ALGORITHMIC TECHNIQUES FOR PREDICTIVE TESTING OF CONCURRENT PROGRAMS AND DISTRIBUTED SYSTEMS

BY

FRANCESCO SORRENTINO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Madhusudan Parthasarathy, Chair and Director of Research
Professor Darko Marinov
Dr. Shaz Qadeer, Microsoft Research
Professor Grigore Roşu

Abstract

The rise of multicore hardware platforms has lead to a new era of computing. In order to take full advantage of the power of multicore processors, developers need to write concurrent code. Unfortunately, as a result of the non-determinism introduced by thread scheduling, multi-threaded programs can show different behaviors even for a single input. Errors in concurrent programs often occur under subtle interleaving patterns that the programmer had not foreseen. There are too many interleavings to explore, even on a fixed test input for a concurrent program, making concurrency testing a hard problem.

Current testing technologies such as stress testing (running the program under test repeatedly with randomized sleep statements and by varying parameters on different platforms) have proved largely inadequate in exposing such subtle interleavings. Among the various techniques to test concurrent programs, the prediction-based technique is one of most the valuable technologies. Starting from one arbitrary concurrent execution of the program under test, alternate interleavings that are likely to contain bugs are predicted. In our research, we explore prediction algorithms based on a combination of static analysis and logical constraint solving to efficiently and effectively test concurrent programs. The strength of our research lies in the fact that the techniques we propose are general enough to predict, with a high degree of accuracy of feasibility, various kinds of concurrency errors. We provide evidence that such an approach is promising in testing concurrent programs. We have implemented our techniques in a framework, PENELOPE. We evaluate it over benchmark programs and find scores of null-pointer dereferences, data-races, atomicity violations and deadlocks by using only a single test run as the prediction seed for each benchmark.

We also take into account the challenge of bringing our experience in predictive testing of concurrent programs to the distributed systems environment. We use supervised machine learning to model the system behaviors in response to perturbations, based on recorded observations in a pseudo-distributed (small-scale) setting. From the learned model, we predict the next system state given current states and applied perturbations. In a perturbation-based testing framework, accurate prediction helps to shorten the waiting time between the consecutive perturbations. Moreover, from the learned model, we reconstruct a possible sequence of perturbations from a given sequence of observed system states for diagnosis. We demonstrate the usefulness of our approach in a case study of a distributed system based on ZOOKEEPER and SOLRCLOUD.

From where I started to where I am today

From what I was to what I became

It has been a long journey.

On the way I lost love but found love too,

I lost friends but found new ones as well,

It has been a tiring journey.

I lost physical closeness of my relatives but found their emotional closeness in return,

I lost a house but found also another one.

Today I'm ready to face a new journey carrying with me the lessons learned in this one,

because in the end ... it is the journey, not the destination, that changes us!

Acknowledgments

First I would like to express my sincere gratitude to my advisor Prof. Madhusudan Parthasarathy for his continual guidance and support throughout the duration of my dissertation work.

I would also like to thank Prof. Darko Marinov, Prof. Grigore Roşu and Shaz Qadeer for serving as members of my dissertation committee and for their insightful comments and feedback during the oral qualifying examination.

I would sincerely thank Prof. Geneva Belford and Prof. Elsa Gunter for their kindness and support.

This dissertation is based on the research I have done with my advisor Prof. Madhusudan Parthasarathy and a number of other collaborators. I would like to thank Prof. Azadeh Farzan and Dr. Niloofar Razavi for the enjoyable collaborations. Some of the ideas developed during these collaborations found their way in published works and finally in this thesis.

I am greatly thankful to Dr. Aarti Gupta and Dr. Malay Ganai for hosting my internship at NEC Labs and for the many inspiring discussions on different topics in testing distributed systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many software applications utilize multi-threaded programming techniques, and as multi-core hardware becomes increasingly powerful and cheaper, the number of such applications further increase. Unfortunately, multi-threaded programming is error-prone due to its high complexity. Concurrency errors –including deadlocks, data-races and atomicity violations– are often difficult to detect because they typically happen under very specific interleavings of the executing threads. One traditional method of testing concurrent programs is *stress testing* repeatedly executes the program under test with the hope that different test executions will result in different interleavings. There are a few problems with this approach. First, the outcome of such testing can be highly dependent on the test environment. For example, some interleavings may only occur in heavily-loaded test systems. Second, this kind of testing depends on the underlying operating system or the virtual machine used for thread scheduling – it does not try to explicitly control the thread schedules; therefore, such testing often ends up executing the same interleaving many times. On the other hand, there are too many interleavings to test and it is impossible to systematically explore all of them, even on a single test input for a concurrent program, making concurrency testing a hard problem.

There are two promising approaches that have emerged in testing concurrent programs: *selective interleavings* and *prediction-based testing* (as well as combinations of them). The selective interleaving approach focuses on testing a *small* but carefully chosen subset of interleavings. There are several tools and techniques that follow this philosophy: for instance, the CHESS tool from Microsoft [70] tests all interleavings that use bounded number preemptions (unforced context-switches), pursuing the belief that most errors can be made to manifest this way. The resulting class of interleavings to consider should be much smaller. This is not quite true. In a program with n threads where each thread executes k steps out of which at most b are potentially blocking, there can be up to $\binom{nk}{(nb+c)!}$ executions when the number of context switches is limited to c [70].

Several tools concentrate on testing *atomicity violating patterns* (for varying notions of what atomicity means), with the philosophy that they are much more likely to contain bugs [61, 74, 75]. However, systematically testing even smaller classes of interleavings is often impossible in practice, as there are often too many of them.

In this work we focus on the *prediction-based testing* methodology. It involves taking one arbitrary concurrent execution of the program under test, and from that predicting alternate interleavings that are likely to contain bugs

(e.g. interleavings that lead to data-races, interleavings that violate atomicity, etc.). Prediction replaces systematic search with a search for interleavings that are *close* to the observed executions only, and hence is more tractable, while at the same time explores interesting interleavings that are likely to lead to errors [29, 30, 89, 91, 98]. We address the challenge of increasing the scalability, the precision and the coverage of such methodology for testing concurrent programs. Scalability is accomplished by taking advantage of static analysis techniques, the precision is obtained by using a logic constraint solver to predict runs, and coverage is improved by introducing a new *relaxed prediction model*. In this work, we also introduce and explore a new target for predictive testing of concurrent programs that is fundamentally different from data-races or atomicity errors: we propose to target executions that lead to *null-pointer dereferences*. The results obtained from our investigations (data-race, atomicity violation, deadlocks and null-pointer dereference predictions) provide evidence of the broad applicability of the proposed approach. The techniques developed are in fact able to predict a large number of feasible runs resulting in bugs on a suite of 18 concurrent benchmarks.

We have brought our experience of predictive testing of concurrent programs to the distributed systems environment as well. We propose perturbation-based learning approaches to model the system behavior using Decision Trees. Once a model is learnt we predict the next system states from given current system states and applied perturbations with a goal to reduce the testing time. We also use the learned system model for diagnosis purpose, where we reconstruct a possible sequence of perturbations from a given sequence of observed system states. We present a case study of an open source system based on ZOOKEEPER and SOLRCLOUD to demonstrate how our techniques can be used to reduce the testing time (we reduced the testing time from 10 hrs to less than 5 hrs), and improve the diagnosis of failures observed in a deployed system (we achieve an accuracy of 59%).

1.1 Research Contributions

The foundation of our research relies on the definition of efficient and effective prediction algorithms for testing concurrent programs. With this in mind, we study the design of prediction algorithms based on a combination of static analysis and logic constraint solving, to test concurrent programs, improving the state of the art. Given an arbitrary execution of a concurrent program under test, we want to accurately and scalably predict executions that are likely to lead to errors.

We implemented a tool PENELOPE that realizes this testing framework and shows that the proposed algorithms are efficient and effective in finding bugs related to different targets[1]. PENELOPE has been released and has attracted the attention of several research groups which are using it in their research project UCLA, USC, UTexas, Indian Institute

[1]Some of the ideas developed in this work are the result of collaborations with my advisor Prof. Madhusudan Parthasarathy, Prof. Azadeh Farzan [40, 91, 41] and Dr. Niloofar Razavi [41]

of Science.

1.1.1 Lock-set Based Approach

A concurrent shared memory program, during its computation, does local computation, reads and writes to shared entities (memory locations), dynamically creates threads, and uses synchronization primitives (such as locks). We exploited a lightweight and fast prediction algorithm that observes only an abstraction of this computation that ignores local computation, ignores precise values read or written, but keeps track precisely of the read and write operations to shared memory locations as well as synchronization primitive usage.

This abstract view of a computation, will be the one that is algorithmically analyzed in order to predict and schedule alternate executions [40, 90, 91]. The prediction algorithm ensures that the predicted run respects the *lock* acquisitions and releases in the run. In other words, the predicted runs are certainly not guaranteed to be feasible in the original program— if the original program had threads that communicated through shared variables in a way that orchestrated the control flow, the predicted runs may not be feasible. Since we can try to execute a predicted interleaving and check whether it is feasible, this will not contribute to the final false positives in a testing scenario.

One of the discoveries brought to light by our experimental evaluation of PENELOPE, while targeting bugs related to atomicity violations and deadlocks, is the effectiveness in finding bugs despite the level of abstraction.

1.1.2 SMT Solver Based Approach

Targeting errors as null-pointer dereferences is particularly challenging because such an error could occur in a thread when dereferencing local variables. Though accurate scalable prediction is intractable, we provide a carefully chosen, novel set of techniques to achieve reasonably accurate and scalable prediction [41]. We use an abstraction to the shared-communication level, take advantage of a static lock-set based pruning, and finally, employ precise and relaxed constraint solving techniques that use an SMT solver to predict schedules.

We evaluate such techniques targeting null-pointer dereferences, data-races and atomicity violations. We find scores of bugs by using only a single test run as the prediction seed for each benchmark. Moreover, we show an improvement in the rate of feasible predicted runs compared to the mere lock-set based approach.

1.1.2.1 Relaxed Logical Prediction

We provide a formulation of the prediction that allows some leeway so that the prediction algorithm can predict runs with mild *deviations* from the observed run which could lead to interesting/buggy runs. This makes the class of predicted runs larger at the expense of possibly making them an *infeasible*, and/or feasible and yet not cause any

error. We mitigate this by using a re-execution engine that accurately executes predicted schedules to check if an error actually occurs [91]. Errors reported are always real and hence we incur no false positives.

1.1.2.2 Pruning

We propose an aggressive pruning of executions using static analysis, based on vector-clocks, that identifies a small segment of the observed run on which the prediction effort can be focused. This greatly improves the scalability of using sophisticated logic solvers. Pruning of executions does not affect feasibility of the runs, but may reduce the number of runs predicted. However, we show that in practice no additional errors were found without pruning.

1.1.3 Case Studies

In order to strengthen our argument, we applied our predictive testing techniques to predict different kinds of errors.

1. **Atomicity violations [91, 40].** A programmer writing a procedure often desires uninterfered access to certain shared data that will enable him/her to think about the procedure locally. The programmer puts together a concurrency control mechanism to ensure this atomicity, often achieved by taking locks associated to the data accessed. This is however extremely error-prone: not acquiring all required locks for the data leads to errors, non-uniform ordering of locking can cause deadlocks, and naive ways of granular locking can inhibit concurrency, which force programmers to invent intricate ways to achieve concurrency and correctness at the same time. When the procedure does non-trivially interact with another concurrent thread in a non-trivial manner, inconsistent state configurations occur and may lead to unexpected behaviors and errors.

2. **Data-races [41].** A data-race occurs when two different threads in a given program can simultaneously access a shared variable, with at least one of the accesses being a write operation. Data-races often lead to unpredictable behavior of programs and are usually symptomatic of errors, especially in programs such as those in our benchmark.

3. **Deadlocks [90].** The simplest mechanism used for synchronizing concurrent accesses to shared data is lock. When threads are not correctly orchestrated deadlock configurations can emerge. A set of threads is deadlocked if each thread deadlocked requests resources, like a lock, held by another thread in the set, which forms a cycle of lock requests.

   We present a fundamentally new predictive approach to detect deadlocks in concurrent programs, and it is not based on cycle detection in lock-graphs. It is based on *lock-sets* and *acquisition histories* (the latter are a kind of hierarchical lock-set information).

4. **Null-pointer dereferences [41].** We propose null-pointer dereferences as a new target for finding bugs in concurrent programs using testing. Such error is very different from data-races, deadlocks or atomicity errors. Null-pointer dereferences can occur in a thread when dereferencing local variables. Consequently, an *accurate* prediction of null-pointer dereferences requires, by definition, handling local variables and the computation of threads. This is in sharp contrast to errors like data-races, deadlocks and atomicity violations, which depend only on accesses to *shared variables*.

1.2 Predictive Testing in the Distributed Systems

We extended the applicability of predictive testing to the context of scalable distributed system. Testing such complex software in a large-scale distributed setting is not only costly but also very challenging. Often severe system-level design defects stay hidden even after deployment and can derail the entire system, possibly resulting in loss of revenue or customer satisfaction.

In order to test such systems, perturbation testing tools have been proposed in which a single perturbation (i.e. act of inducing controlled changes to the execution) is applied to a system-under-test (SUT) at a time. Before the next perturbation is applied, the test engine waits until the effect of the applied perturbation is 'detected' (by anomaly detection code), and 'absorbed' (by the anomaly management code through corresponding actions). Note, after applying each perturbation – such as a leader node failure –, the system reacts, i.e., it goes through internal changes, such as a new leader election process.

We propose a framework to address two different aspects of testing scalable distributed systems [92][2]. We present a case study of an open source system based on ZOOKEEPER [15] and SOLRCLOUD [2] to demonstrate how our techniques can be used to reduce the testing time (we reduced the testing time from 10 hrs to less than 5 hrs), and improve the diagnosis of failures observed in a deployed system (we achieve an accuracy of 59%).

1.2.1 Testing Problem

These perturbation testing tools, e.g., SETSUDŌ, use the conservative approach of waiting a reasonable amount of time so that the SUT has completely reacted to the perturbation applied (i.e., reached a steady state). In general, such a conservative approach (i.e., waiting to ensure there is no further log update) is quite time consuming, especially, when one needs to apply millions of perturbations. One can reduce the wait time by knowing exactly when the system has absorbed the changes. This requires heavy-weight instrumentation which will incur performance overhead, resulting in increased testing time instead of reducing it. Informally, we refer to this issue as a qualitative testing problem, i.e.,

[2]This part of the work was started and developed during my internship at NEC Labs.

can we reduce the testing time without incurring performance overhead?

We use supervised machine learning (based on decision tree classifiers) to model the system behaviors in response to perturbations, based on recorded observations in a pseudo-distributed (i.e., distributed but not at a very large scale) setting. From the learned model, we predict the next system state given current states and applied perturbations. In a perturbation-based testing framework, accurate prediction helps to shorten the waiting time between the consecutive perturbations.

1.2.2 Diagnosis Problem

Generally the effects of environmental perturbations to the running system (i.e., reactions of the system to the perturbations) are logged. Given a sequence of recorded system states prior to the observed system defects, it may be useful to know what the environmental perturbations were that triggered the system failure. Knowing such triggers often helps to recreate the defect. Informally, we refer this problem as a diagnosis problem, i.e., *can we predict a possible perturbation sequence from a given sequence of system states?*

From the learned model, we reconstruct a possible sequence of perturbation from a given sequence of observed system states for diagnosis.

1.3 Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2 we review background information and related work that is relevant to the rest of the thesis. In Chapter 3 we give an overview of the prediction-based approach. In Chapter 4 we present the lightweight prediction algorithms based on lock-set and acquisition history for the prediction of atomicity violations and deadlocks. In Chapter 5 we present the precise and the relaxed prediction algorithms for the prediction of null-pointer dereferences, atomicity violations and data-races. In Chapter 6 we provide the implementation details of our framework PENELOPE. In Chapter 7 we evaluate the result obtained by PENELOPE against different benchmarks. In Chapter 8 we present a new application of prediction-based testing on distributed systems. Finally, Chapter 9 concludes the thesis and presents potential future directions for our research.

Chapter 2

Background and Related Work

This Chapter provides useful context for understanding the material in the following Chapters. In particular, the formal notation to describe predicted runs and details on the most common bugs that afflict concurrent programs; therefore targets of our study. Finally, it provides an overview of the related work in the area of testing concurrent programs.

2.1 Modeling Program Runs

We model the runs of a concurrent program as a *word* where each letter describes the action done by a thread in the system. The word will capture the essentials of the run— shared variable accesses, synchronizations, thread-creating events, etc. However, we will *suppress* the local computation of each thread, i.e. actions a thread does by manipulating local variables, etc. that are not (yet) visible to other threads, and model the local computation as a *single* event *lc*. In the formal treatment, we will ignore other concurrency constructs such as barriers, etc.; these can be accommodated easily into our framework.

We fix an infinite countable set of thread identifiers $\mathcal{T}$= $\{T_1, T_2, \ldots\}$ and define an infinite countable set of shared variable names SV that the threads manipulate. Without loss of generality, we assume that each thread T_i has a *single* local variable lv_i that reflects its entire local state. Let V = $SV \bigcup_i \{lv_i\}$ represent the set of all variables. Let *Val*(x) represent the set of possible values that variable $x \in SV$ can get, and define *Init*(x) as the initial value of x. We also fix a countable infinite set of locks $\mathcal{L}$=$\{l_1, l_2, \ldots\}$.

The actions that a thread $T_i \in \mathcal{T}$ can perform on a set of shared variables SV and global locks $\mathcal{L}$ is defined as:

$$\Sigma_{T_i} = \{T_i\text{:}\,\textit{read}_{x,val},\ T_i\text{:}\,\textit{write}_{x,val} |\ x \in SV, val \in \textit{Val}(x)\} \cup \{\ T_i\text{:}\,\textit{lc}\}$$
$$\cup\, \{T_i\text{:}\,\textit{acquire}(l),\ T_i\text{:}\,\textit{release}(l) |\ l \in \mathcal{L}\} \cup \{T_i\text{:}\ tc\ T_j \mid T_j \in \mathcal{T}\}$$

Actions T_i:*read*$_{x,val}$ and T_i:*write*$_{x,val}$ correspond to the thread T_i reading the value val from and writing the value val to the shared variable x, respectively. Action T_i: *lc* corresponds to a local computation of thread T_i that accesses and changes the local state lv_i. Action $T_i\colon acquire(l)$ represents acquiring the lock l and the action $T_i\colon release(l)$

represents releasing of the lock l, by thread T_i. Finally, the action T_i: $tc\ T_j$ denotes the thread T_i creating the thread T_j.

We define $\Sigma = \bigcup_{T_i \in \mathcal{T}} \Sigma_{T_i}$ as the set of actions of all threads.

2.1.1 Modeling Atomic Execution Blocks

In order to target atomicity violations in our testing algorithms we assume that the program is annotated with begin-block and end-block annotations; a block thus demarcated *execution block* in code. These execution blocks correspond to code blocks that may have been reasoned with locally, without considering interleavings from other threads.

In practice, we will choose such an annotation automatically, typically by marking methods and certain loop-bodies as execution blocks. Note that these annotations have no semantic value– in particular, they are certainly not respected by the compiler in any way. These transactional boundaries are also part of the execution that we observe.

Hence, given an execution, each thread executes in it a series of *transactions*. A transaction is a sequence of actions; each action can be a read or a write to a (global) variable, or a synchronization action.

Let us now define the notation to talk about blocks. The actions Σ_{T_i}, that a thread T_i can perform defined in 2.1 is now enriched with $\{T_i{:}\,\triangleright, T_i{:}\,\triangleleft\}$. Actions $T_i{:}\,\triangleright$ and $T_i{:}\,\triangleleft$ correspond to the beginning and the end of execution blocks in thread T_i, respectively.

2.2 Semantically and Syntactically Valid Runs

Notation: For any alphabet A, and any word $w \in A^\star$, let $w[i]$ ($1 \leq i \in |w|$) denote the letter in the i'th position of w, and $w[i,j]$ denote the substring $w[i]w[i+1]\ldots w[j]$ of w. For $w \in A^\star$ and $B \subseteq A$, let $w|_B$ denote the word w projected to the letters in B.

A word w in $\Sigma^\star$, in order to represent a run, must satisfy several obvious semantic and syntactic restrictions, which are defined below.

2.2.1 Lock-validity, Data-validity, and Creation-validity

There are, obviously, certain semantic restrictions that a run must follow. In particular, it should respect the semantics of locks and semantics of reads, i.e. whenever a read of a value from a variable occurs, the last write to the same variable must have written the same value, and the semantics of thread creation. These are captured by the following three definitions.

Definition 2.2.1 (Lock-validity) *A run $\rho \in \Sigma^*$ is* lock-valid *if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T_i\!:\! acquire(l), T_i\!:\! release(l) \mid T_i \in \mathcal{T}\}$ denote the set of locking actions on lock l. Then ρ is lock-valid if for every $l \in \mathcal{L}$, $\rho|_{\Sigma_l}$ is a prefix of*

$$\left[\bigcup_{T_i \in T}(T_i\!:\! acquire(l)\ T_i\!:\! release(l))\right]^* \qquad \blacksquare$$

Definition 2.2.2 (Data-validity) *A run $\rho \in \Sigma^*$ over a set of threads $\mathcal{T}$, variables SV, and locks $\mathcal{L}$, is* data-valid *if it respects the read-write constraints. Formally, for each n such that $\rho[n] = T_i\!:\! read_{x,val}$, one of the following holds:*
(i) The last write action to x writes the value val. I.e. there is a $m < n$ such that $\rho[m] = T_j\!:\! write_{x,val}$ and there is no $m < k < n$ such that $\rho[k] = T_q\!:\! write_{x,val'}$ for any val' and any thread T_q, or
(ii) there is no write action to variable x before the read, and val is the initial value of x. I.e. there is no $m < n$ such that $\rho[m] = T_j\!:\! write_{x,val'}$ (for any val' and any thread T_j), and $val = Init(x)$. $\blacksquare$

Definition 2.2.3 (Creation-validity) *A run $\rho \in \Sigma^*$ over a set of threads $\mathcal{T}$ is* creation-valid *if every thread is created at most once and its events happen after this creation, i.e., for every $T_i \in \mathcal{T}$, there is at most one occurrence of the form $T_j\colon tc\ T_i$ in w, and, if there is such an occurrence, then all occurrences of letters of Σ_{T_i} happen after this occurrence.* $\blacksquare$

2.2.2 Block-validity

A word in Σ^*, in order to represent a run when atomic blocks are declared in the program, must satisfy a few obvious restrictions, captured by the following definition. Define a sequence of execution blocks in thread $T_i \in \mathcal{T}$ as:

$$ExecBlk_{T_i} = ((T_i\!:\!\triangleright).[(T_i\!:\! lc\ GlobComp_i)^* T_i\!:\! lc].(T_i\!:\!\triangleleft))^*$$

where $GlobComp_i = \{T_i : read_{x,val}, T_i : write_{x,val} \mid x \in SV \text{ and } val \in Val(x)\} \cup \{T_i : acquire(l),\ T_i : release(l) \mid l \in \mathcal{L}\}$. A sequence of execution blocks in a thread T_i is a word which has properly delineated blocks, and where between every two global actions (and between a block demarcation and a global action) there is precisely one event $T_i : lc$.

Definition 2.2.4 (Block-validity) *A run $\rho \in \Sigma^*$ over a set of threads $\mathcal{T}$, variables SV, and locks $\mathcal{L}$, is* block-valid *if for each thread $T_i \in \mathcal{T}$, $\rho|_{T_i}$ is a prefix of $ExecBlk_{T_i}$.*

Given an execution ρ we would like to *infer* other executions ρ' from ρ. An execution ρ' belongs to the inferred set of ρ iff ρ'_T is a prefix of ρ_T, for every thread T. Moreover ρ' will be lock-valid, creation-valid, data-valid, block-valid or a combination of them, depending on the precision we want to achieve in our prediction (see Chapters 4 and 5). Let $Infer(\rho)$ denote the set of executions inferred from ρ.

2.2.3 Program Order

The occurrence of actions in runs are referred to as *events* in this work. Formally, the set of events of the run is $E = \{e_1, \ldots, e_n\}$ and an event in a set of local executions $\{\rho_T\}_{T \in \mathcal{T}}$ is a pair (T, i) where $T \in \mathcal{T}$ and $1 \leq i \leq |\rho_T|$. In other words, an event is a particular action one of the threads executes.

Let ρ be a global execution, and $e = (T, i)$ be an event in $\{\rho_T\}_{T \in \mathcal{T}}$. Then we say that the j'th action ($1 \leq j \leq |\rho|$) in ρ *is the event e* (or, $Event(\rho[j]) = e = (T, i)$), if $\rho[j] = T{:}a$ (for some action a) and $\rho_T[1, i] = \rho[1, j]|_T$. In other words, the event $e = (T, i)$ appears at the position j in ρ in the particular interleaving of the threads that constitutes ρ. Reversely, for any event e in $\{\rho_T\}_{T \in \mathcal{T}}$, let $Occur(e, \rho)$ denote the (unique) j ($1 \leq j \leq |\rho|$) such that the j'th action in ρ is the event e, i.e $Event(\rho[j]) = e$. Therefore, we have $Event(\rho[Occur(e, \rho)]) = e$, and $Occur(Event(\rho[j])) = j$.

While the run ρ defines a total order on the set of events in it $(E, \leq)$, there is an induced total order between the events of each thread. We formally define this as $\sqsubseteq_i$ for each thread T_i, as follows: for any $e_s, e_t \in E$, if a_s and a_t belong to thread T_i and $s \leq t$ then $e_s \sqsubseteq_i e_t$. The partial order that is the union of all the program orders is $\sqsubseteq = \cup_{T_i \in T} \sqsubseteq_i$.

2.3 Lock-set and Acquisition History

Let ρ be an execution and let $\{\rho_T\}_{T \in \mathcal{T}}$ be its set of local executions. Consider ρ_T (for any T). The *lock-set held after* ρ_T is the set of all locks T holds: $LockSet(\rho_T) = \{l \in \mathcal{L} \mid \exists i.\rho_T[i] = T{:}acquire(l)$ *and there is no* $j > i$ *and* $\rho_T[j] = T{:}release(l)\}$.

The *acquisition history* of ρ_T records, for each lock l held by T at the end of ρ_T, the set of locks that T acquired (and possibly released) by T after the last acquisition of the lock l. Formally, the acquisition history of ρ_T, $AH(\rho_T) : LockSet(\rho_T) \rightarrow 2^{\mathcal{L}}$, where $AH(l)$ is the set of all locks $l' \in \mathcal{L}$ such that $\exists i.\rho_T[i] = T{:}acquire(l)$ and there is no $j > i$ such that $\rho_T[j] = T{:}release(l)$ and $\exists k > i.\rho_T[k] = T{:}acquire(l')$.

Consider the following program structure:

```
synchronize (l1) {
    // point A
    synchronize (l2) {
        // point B
    }
    synchronize (l3) {
        // point C
    }
}
```

We have

$$\begin{aligned} LockSet(A) &= \{l_1\}, \\ LockSet(B) &= \{l_1, l_2\} \\ LockSet(C) &= \{l_1, l_3\}. \end{aligned}$$

Also, we have $AH(A) = \{(l_1, \{\})\}$, $AH(B) = \{(l_1, \{l_2\}), (l_2, \{\})\}$, and $AH(C) = \{(l_1, \{l_2, l_3\}), (l_3, \{\})\}$. Two acquisition histories AH_1 and AH_2 are said to be *compatible* if there are no two locks l and l' such that $l' \in AH_1(l)$ and $l \in AH_2(l')$. They are otherwise said to be *not compatible*.

It will hereafter be clear that acquisition histories give not only enough traction to detect alternate interleavings, but also provide an effective mechanism to predict deadlocks and to *re-schedule* predicted interleavings.

2.3.1 Nested-locking and Pairwise Reachability

A (global) execution ρ over $\mathcal{T}$ and $\mathcal{L}$ is said to respect *nested-locking* if there is no thread T and two locks l and l' such that $\rho|_{\Pi_{\{l,l'\},\{T\}}}$ has a contiguous subsequence $T\ acquire(l) T\ acquire(l') T\ release(l)$. Where $\Pi_{\{l,l'\},\{T\}} = \{T\ acquire(l), T\ release(l), T\ acquire(l'), T\ release(l') |\ l, l' \in \mathcal{L}\}$. In other words, an execution respects nested-locking if each thread releases locks strictly in the reverse order in which they were acquired.

At the level of abstraction we observe executions (namely observing reads, writes, and lock acquisitions and releases), it turns out that we can precisely capture when two sequences of events of threads T_1 and T_2 can be combined to a lock-valid execution using just lock-sets and acquisition histories. More precisely, let ρ be an execution and let w_1 be a prefix of $\rho|_{T_1}$ and w_2 be a prefix of $\rho|_{T_2}$. Then there is an execution ρ' such that $\rho'|_{T_1} = w_1$ and $\rho'_{T_2} = w_2$ *if and only if* the lock-sets of T_1 and T_2 after w_1 and w_2 are disjoint and the acquisition histories at the end of w_1 and w_2 are compatible (this technical result is due to Kahlon et al. [58]). This result will underpin our algorithms for predicting and scheduling, as it accurately captures those schedules that are feasible in a abstract program where only locks semantic is respected (i.e. data flow ignored).

Kahlon et al. [58] argues that global reachability of two threads communicating via nested locks is effectively and *compositionally* solvable by extracting locking information from the two threads in terms of *acquisition histories*.

In particular, there is an execution that ends with event e in one thread and event f in the other thread, if, and only if, the acquisition history at e and that at f are compatible.

Lemma 2.3.1 (Kahlon et al. [58]) *Let ρ be an execution of a concurrent program P, let $\{\rho_T\}_{T \in \mathcal{T}}$ be its set of local executions, and let T_1 and T_2 be two different threads. Let $e = (T_1, i)$ be an event of thread T_1 and $f = (T_2, j)$ be an event of thread T_2 of these local executions.*

There is a run $\rho' \in Infer(\rho)$ such that e and f occur in ρ', and further, $\rho'_{T_1} = \rho_{T_1}[1, i]$ and $\rho'_{T_2} = \rho_{T_2}[1, j]$

if, and only if,

$LockSet(\rho_{T_1}[1,i]) \cap LockSet(\rho_{T_2}[1,j]) = \emptyset$, and the acquisition history of $\rho_{T_1}[1,i]$ and the acquisition history of $\rho_{T_2}[1,j]$ are compatible. ∎

2.4 Common Bugs and Error Patterns in Concurrent Programs

We applied our predictive testing techniques to predict different errors as data-races, deadlocks and null-pointer dereferences. We also applied out algorithms to the prediction of potential buggy patterns like atomicity violations. In the rest of the Section we will describe these targets.

2.4.1 Atomicity Violations

A programmer writing a procedure often desires uninterfered access to certain shared data that will enable him/her to reason about the procedure locally. The programmer puts together a concurrency control mechanism to ensure this atomicity, often achieved by taking locks associated to the data accessed. This is however extremely error-prone: not acquiring all required locks for the data leads to errors, non-uniform ordering of locking can cause deadlocks, and naive ways of granular locking can inhibit concurrency, which force programmers to invent intricate ways to achieve concurrency and correctness at the same time. When the procedure does non-trivially interact with another concurrent thread in a non-trivial manner, inconsistent state configurations occur and may lead to unexpected behaviors and errors.

We now define atomicity as the notion of *conflict serializability*. Define the *dependency* relation D as a symmetric relation defined over the events in Σ, which captures the dependency between (a) two events accessing the same entity, where one of them is a write, and (b) any two events of the same thread, i.e.,

$$\begin{aligned} D \quad = \quad & \{(T_1{:}a_1, T_2{:}a_2) \mid T_1 = T_2 \textit{ and } a_1, a_2 \in \Sigma \textit{ or} \\ & \exists x \in SV \textit{ such that } (a_1 = \text{read}(x) \textit{ and } a_2 = \text{write}(x)) \textit{ or} \\ & (a_1 = \text{write}(x) \textit{ and } a_2 = \text{read}(x)) \textit{ or } (a_1 = \text{write}(x) \textit{ and } a_2 = \text{write}(x))\} \end{aligned}$$

Definition 2.4.1 (Equivalence of runs) *The equivalence of runs is defined as the* smallest *equivalence relation $\sim \subseteq \Sigma^\star \times \Sigma^\star$ such that: if $\rho = \rho ab \rho'$, $\rho' = \rho ba \rho' \in \Sigma^\star$ with $(a,b) \notin D$, then $\rho \sim \rho'$.*

It is easy to see that the above notion is well-defined. Two runs are considered equivalent if we can derive one run from the other by iteratively swapping consecutive independent actions in the run.

We call a run ρ *serial* if all the transactions in it occur sequentially: formally, for every i, if $\rho[i] = T{:}a$ where $T \in \mathcal{T}$ and $a \in \Sigma$, then there is some $j < i$ such that $T[i] = T{:}\triangleright$ and every $j < j' < i$ is such that $\rho[j'] \in \Sigma_T$. In other

words, the run is made up of a sequence of complete transactions from different threads, interleaved at boundaries only.

Definition 2.4.2 *A run is* serializable *(or atomic) if it has an equivalent serial run. That is, ρ is a serializable run if there is a* serial *run ρ' such that $\rho \sim \rho'$.*

2.4.1.1 Minimal Serializability Violations

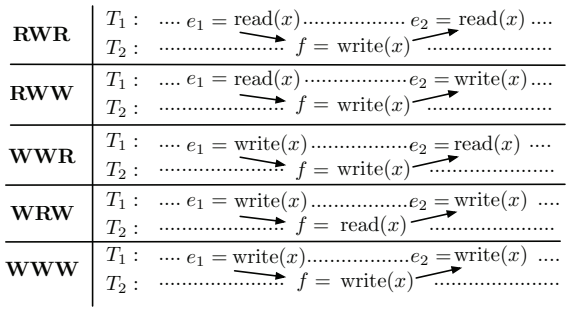
In recent work [39], it has been studied the meta-analysis problem for atomicity, and shown that when all synchronization actions in the execution are *ignored*, efficient algorithms that work in time *linear* in n (where n is the length of the execution) are feasible. However, we also showed that if the locking synchronization between threads is taken into account, an algorithm that is linear in n is *unlikely* to exist.

We show that when the program uses only *nested locking* (i.e. when threads release locks in the reverse order of how they acquired them), we can build algorithms that effectively analyze *all* interleavings for basic atomicity violations in time that is *linear* in the length of the execution.

In order to find an efficient scalable algorithm for analyzing executions for atomicity violations we study only atomicity violations caused by two threads accessing one variable only. More precisely, we look for *minimal serializability* violations which are those caused by two threads and one variable only; i.e. we look for threads T_1 and T_2, where there are two events e_1 and e_2 that access a variable v in a single transaction of T_1, and there is an action in thread T_2 that is happens in between the events e_1 and e_2, and is conflicting with both e_1 and e_2.

In our experience in studying concurrency bugs, we have found that many atomicity violations are caused due to such patterns of interaction. The recent study of concurrency bugs by Lu et al. [66] in fact found that 96% of concurrency errors involved only two threads, and 66% of (non-deadlock) concurrency errors involved only one variable. The restriction to atomicity errors involving only two threads and one variable makes our algorithm feasible in practice.

There are *five* access patterns that correspond to simple atomicity (serializability) violations of *two threads* and *one variable*. A pattern consists of two events e_1 and e_2 which belong to a same execution block of a thread T_1, and a third event f which belongs to another thread T_2. These events should appear in a run ρ such that f occurs after e_1, and e_2 occurs after f in ρ, i.e. $\rho = \ldots e_1 \ldots f \ldots e_2 \ldots$. Moreover, e_1, f, and e_2 should be all accesses to the same shared variable x, and f should have conflict with both e_1 and e_2. Two events have conflict when at least one of them is a write access. As a result, a pattern should be of one the following formats:



2.4.2 Data-races

A data-race occurs when two concurrent threads access a shared variable and when:

- at least one access is a write
- the threads use no explicit mechanism to prevent the accesses from being simultaneous

If a program has a potential data-race, then the effect of the conflicting accesses to the shared variable will depend on the interleaving of the thread executions. Although programmers occasionally deliberately allow a data-race when the nondeterminism seems harmless, usually a potential data-race is a serious error caused by failure to synchronize properly.

2.4.3 Null-pointer Dereferences

We target executions that lead to *null-pointer dereferences*. Such error is very different from data-races or atomicity errors. Null-pointer dereferences can occur in a thread when dereferencing local variables. Consequently, an *accurate* prediction of null-pointer dereferences requires, by definition, handling local variables and the computation of threads. This is in sharp contrast to errors like data-races and atomicity violations, which depend only on accesses to *shared variables*.

2.4.4 Deadlocks

Deadlocks in a shared-memory concurrent program are unintended conditions that can be mainly classified into two types: *resource-deadlocks* and *communication deadlocks*. A set of threads is resource-deadlocked if each thread deadlocked is waiting for a resource, like a lock, held by another thread in the set, which forms a cycle of lock requests. In communication deadlocks, some threads wait for messages that do not get sent because the sender threads are blocked or they have already sent the messages before receiving threads start to wait. In [55] the authors illustrate

that it could be really hard to precisely detect all kinds of deadlocks by the same techniques. In this study we focused only on resource deadlocks, from now on referred to as deadlocks.

2.5 Related Work

We compare our proposed approach and framework with related work in different categories.

2.5.1 Selective Testing

Considering *selective interleavings* as approach to test concurrent program, several tools and techniques have been proposed: for example CHESS[70], from Microsoft systematically exercises all executions that involve only a few context-switches, guided by the intuition that many errors manifest with a few context-switches/preemptions. However, CHESS is constrained by the number of executions it must explore, which even with two context-switches grows quadratically in the length of the executions, making it infeasible for long tests. Some of the example executions handled by PENELOPE in this framework overwhelm CHESS. On the other hand, CHESS has one distinct advantage over execution-based analysis techniques (including PENELOPE) in that it explores *all* executions with two context-switches and is not constrained by the events that occur in a particular observed schedule. We believe that PENELOPE and CHESS are complimentary in the their coverage of the interleaving space.

2.5.2 Prediction-based Testing

2.5.2.1 Logic-based Methods

Logic-based prediction approaches, target precise prediction. The closest work related to ours in the realm of logic-based methods are those that stem from *bounded model-checking* for finding executions of *bounded length* in concurrent programs that have bugs. Typically, a program's loops are unrolled a few times to get a bounded program, and using a logical encoding of the runs in this bounded program, a constraint solver is used to check if there is an error. We refer the reader to the papers from the NEC Labs group [87, 88] as well as work from Microsoft Research [1, 36, 60, 78], where the programs are converted first to a sequential program from which bounded run constraints are generated.

Given an execution of the program, several work [96, 97] model the whole computation (local as well as global) logically to guarantee feasibility. The research presented in the above related work has too big an overhead to scale to large executions.

2.5.2.2 Other Methods

There are several work on prediction-based testing that do not use logical methods. Those by Wang and Stoller [98, 99], and [51] by Huang and Zhang. A more liberal notion of generalized dynamic analysis of a single run has also been studied in a series of papers by Chen et al. [29, 30] and Meredith et al. [69]. JPREDICTOR [30] offers a predictive runtime analysis that uses *sliced causality* [29] to exclude the irrelevant causal dependencies from an observed run and then exhaustively investigates all of the interleavings consistent with the sliced causality to detect potential errors. RV [69] combines sliced causality, with lock-atomicity. As opposed to our approach, JPREDICTOR and RV cannot predict runs causally different from the observed run and the precision of the predictions depends on the static analysis performed to obtain sliced causality.

In general, the main drawback of the non-logical prediction approaches is that they mostly suffer from imprecision; i.e. the predicted runs may not be feasible. In fact, they ignore data which makes them less effective in finding bugs that are data-dependent such as null-pointer dereferences. Maximal Causality Model (MCM) [85], allows prediction at the level of shared communication and is the most precise prediction one can achieve at this level. In particular, we use this model to predict runs leading to null-pointer dereferences.

2.6 Bugs Targeted

2.6.1 Null-pointer Dereference

A closely related work is CONMEM [108] (see also [107]), where the authors target a variety of memory errors in testing concurrent programs, including null-pointer dereferences, but the prediction algorithms are much weaker and quite inaccurate compared to our robust prediction techniques. Furthermore, there is no accurate rescheduling engine which leads the tool to have many false positives.

2.6.2 Atomicity Violations

There are two main streams of work that use predictive analysis for detecting atomicity violations in concurrent programs that are relevant. In two papers [98, 99], Wang and Stoller study the prediction of runs that violate serializability from a single run, similar to our prediction algorithm. The algorithm we propose in Chapter 4 [40] (and implemented as a module of PENELOPE) is better as it works with little memory overhead, while that of Wang and Stoller keeps track of a large graph, which doesn't scale as the size of executions increases. Note that while both these tools predict possible atomicity violations, they do not actually synthesize alternate schedules nor try to reschedule the program to expose these errors, and have a lot of false-positives. In [97], the prediction algorithm of [40] in combination with

SMT solvers was used to remove false positives.

A recent work related to ours is the tool CTRIGGER [75], that has similar motivation as ours in finding and scheduling atomicity violations. CTRIGGER works by examining a few executions, finding points where atomicity violations could occur, prunes away many schedules that are infeasible due to mutually excluded blocks or ordering constraints, and tries to schedule a selected subset of the rest by scheduling those that are less likely to manifest. The algorithms for pruning are however entirely based on heuristics, with little algorithmic analysis, unlike PENELOPE.

There has also been work on finding atomicity violations by using a *generalized* dynamic analysis of an execution. SIDETRACK is a tool [106] that finds atomicity violations by a generalized analysis of the observed run. Note that this technique does not examine runs that are causally different (as PENELOPE does), and hence does not do any rescheduling. Moreover, this technique only detects and reports atomicity violations for a programmer to examine, and cannot produce error traces that violate the test harness (in fact, a generalized run will violate the harness if and only if the original run already violated the harness).

A more liberal notion of generalized dynamic analysis of a single run have also been studied in a series of papers by Rosu et al. [30, 83], who use static analysis of the program to build a causal map to analyze for atomicity.

The run-time *monitoring* algorithms and tools for detecting violations of different types in just the observed run are well understood. The tool by Farzan and Madhusudan reported in [38] provides the most space-efficient algorithm known for the atomicity violations monitoring problem, as the space overhead is bounded when the number of threads and variables are fixed (the algorithm used in VELODROME[44] also can perform sound and complete serializability detection, but the space-overhead is not bounded). These techniques work for *any number of threads and variables*. JAVA-MOP[52] provides an infrastructure for combining formal specification and implementation by automatic generation of monitoring code and instrumentation of monitored programs for Java. Monitoring, of course, is a much simpler problem than prediction.

AVIO [67] defines *access interleaving invariants*— certain patterns of access interactions on variables— and learns the intended access interleaving using tests, and monitors runs to find errors. A variant of dynamic two-phase locking algorithm [73] for detection of serializability violations is used in the atomicity monitoring tool developed in [102].

There has also been recent work on *active randomized testing* resulting in tools as ATOMFUZZER [74], ASSET-FUZZER [61]. These tools execute and "hold" threads at strategic points to try to manifest errors. The *randomness* approach was initially proposed in CONTEST [16], a concurrency testing tool for java programs, from IBM. It attempts to insert noise (randomness) at various synchronization operations while dynamically running the program. It uses a variety of heuristics to drive the schedules. It closely resembles random walk in both behavior and technique.

We found that the holding of threads dynamically and randomly at strategic points are a bit unpredictable in our experience with the tool. This technique also does not have the capability of handling nested locks, and interrupts

threads at wrong positions which lead it to not find errors.

Apart from the related work discussed above, *atomicity violations based on serializability* have been suggested to be effective in finding concurrency bugs in many works [42, 43, 102]. Lipton transactions have been used to find atomicity violations in programs [42, 43, 49, 65].

2.6.3 Data-races

MCM [85] has been used by Said et al. [80] for finding data-race witnesses. There has also been recent work on *active race directed testing* resulting in a tool RACEFUZZER [84]. Such tools is composed by two phases. RACEFUZZER uses an existing hybrid dynamic race detection to compute a pair of program states that could potentially result in a race condition. Then, it randomly selects a thread schedule until a thread reaches one program location that is part of the potential data-race pair. At this point the execution of the thread is paused at that program location, while the other thread schedules are randomly picked in order to drive another thread to the second program location.

2.6.4 Deadlocks

Prior work on deadlock detection in concurrent programs have exploited different techniques: dynamic (including postmortem) analysis, model checking and static analysis.

Static approaches attempt to detect possible deadlocks directly on the source code and do not require the execution of the application being tested [21, 37, 71, 86, 101]. Even if this approach exhaustively explores all potential deadlocks, it suffers from high false positives, aggravating the user. For example, Williams et al. in [101] report that on 1,000 potential deadlocks only 7 were real deadlocks. In order to reduce the number of false positives numerous directions have been explored. Williams et al. [101] have used heuristics to try to remove some of the false positives but these have the potential of removing some real deadlocks. von Praun [95] uses a context-sensitive lock-set and a lock graph in his approach. To reduce false positives they suppress certain deadlocks based on lock alias set information, again potentially removing real deadlocks. `RacerX` [37] is a static data race and deadlock detection tool for concurrent programs written in C. Additional programmer's annotations are used to inject the programmer's intent and consequently suppress false positives and improve the `RacerX`'s accuracy. More recently, Naik et al. [71] combine a suite of static analysis techniques to cut the false positive rates. Unfortunately, scalability and problems related to conditional statements still remain a drawback of static analysis.

Several researchers have explored a model-checking approach to detect deadlocks in concurrent programs using model checker such as `SPIN` and `Java Pathfinder` [34, 50, 93]. Joshi et al. [55] monitor the annotated conditional variables as well as lock synchronization and threading operations in a program to generate a trace program containing not only thread and lock operations but also the value of conditionals. Then they apply `Java`

`Pathfinder` to check all abstracted and inferred execution paths of the trace program to detect deadlocks. However, the technique proposed requires manual effort to design and add annotations, which can be error-prone, and suffers from the scalability issue to handle large-scale programs. Dynamic analysis techniques have been extensively explored as well [20, 23, 24]. With this approach the execution of one or more runs of a concurrent program are monitored to determine if a deadlock may occur in another execution of the program. Eraser [82] is a dynamic analysis program originally designed to detect possible data-races and later modified to detect possible deadlock conditions. Eraser is neither a sound nor complete algorithm. Bensalem et al. [23, 24] find potential deadlocks in Java programs by finding cycles in the lock graph generated during program execution. All cycles in the lock graph are considered to be potential deadlocks. This causes a lot of false positives. They use the happen-before relation to improve the precision of cycle detection and use a guided scheduler to confirm a deadlock. Farchi et al. [20] proposed an approach where they generate a lock graph across multiple runs of the code. Deadlocks prediction is done searching cycles in this graph; unfortunately, this approach may also produce false alarms. `MulticoreSDK` [68] and `DeadlockFuzzer` [56] use lock-set based strategies to predict potential deadlocks. Once a potential deadlock has been found, deadlock confirmation, avoidance, or healing strategies can be applied. Neither approaches are capable of completing large executions, moreover the rescheduling phase is not robust enough to guarantee that the right time to trigger a deadlock is used.

`MagicFuzzer` [27] is a dynamic resource deadlock detection technique based on locks dependencies. It locates potential deadlock cycle from an execution, it iteratively prunes lock dependencies that have no incoming or outgoing relations with other lock dependencies. Similarly to our approach it has a trace recording phase, a potential deadlock detection phase and a deadlock confirmation phase — which avoids false positive. However, like most of the techniques based on cycle detection, the detection phase is not precise, even assuming that the communication between threads occurs only through locks, overloading the confirmation phase.

Finally, some techniques have been proposed to prevent deadlock from happening during the execution, and to recover from deadlocks during execution. `Deadlock Immunity` prevents the re-occurrence of a deadlock using an online monitoring algorithm and maintaining a database containing all patterns of occurred deadlock [57]. It does not have an active re-scheduling or potential deadlock detection component. `Gadara` statically detects deadlocks and use a gate locks based system to avoid their occurrence. However, it may suffer of both false positives and negatives when detecting deadlocks [100].

Chapter 3

The Prediction Methodology

Multi-threaded programs may exhibit different behaviors when executed at different time, even with a fixed single input. This intrinsic nondeterminism makes multi-threaded program difficult to analyze, test and debug. This work presents predictive analysis techniques to detect concurrency errors from observing execution traces of the the program under test. The techniques we propose are sound, *no false alarms* are generated. The program under test is automatically instrumented to record runtime events. The execution that is observed needs not hit the error, yet, errors in other execution can be correctly predicted.

Previous efforts tend to focus on either soundness or coverage. For example methods based on happen-before relation try to be sound, but have a limited coverage over interleavings, thus missing errors. On the other hand, methods based on lock-set have a better coverage but suffer from false positive.

Our analysis techniques proposed in this work aim at improving the coverage without giving up on soundness meanwhile remaining general and applicable to the prediction of different targets. The coverage is improved by leveraging on the lock-set based approach and a novel analysis: relaxed prediction. The soundness is achieved through the rescheduling phase and the SMT solver based approach.

3.1 Predictive Runtime Analysis

Concurrency related programming errors are due to incorrectly constrained interactions of the concurrent threads. Regardless their variety of manifestations these bugs can be divided in two categories. Bugs due to under-constraining and bugs due to over-constraining. Bugs in the first category are those in which the threads have more freedom in interacting with other threads than they should have, leading to data-race, atomicity violations, order violations. Bugs in the second category are those in which the threads are more restricted than they should be, leading to either deadlocks or performance bugs.

However, all these bugs share the scheduling sensitivity, and the often humongous number of possible thread interleavings exacerbates their detection. Bugs manifest themselves rarely during the program execution.

Runtime concurrency bug detection and prediction come mainly in two flavors. When the target is to detect runtime

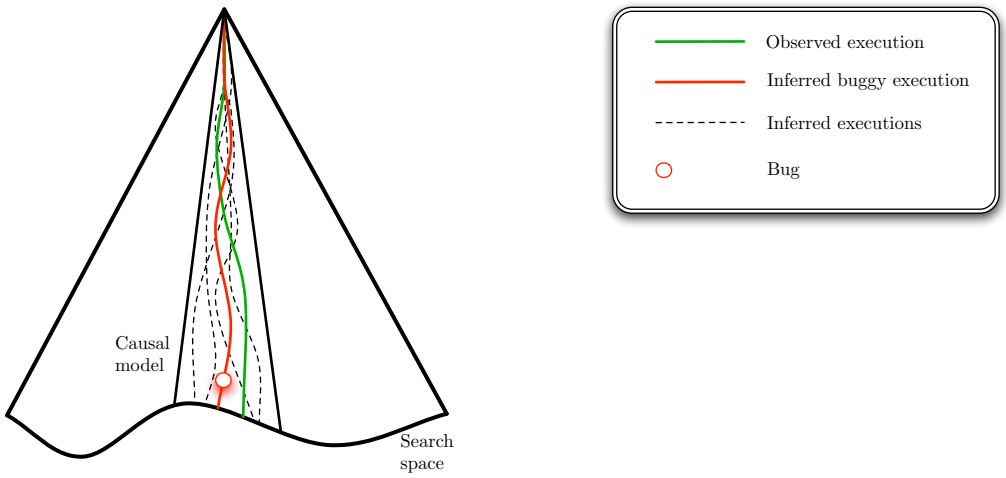


Figure 3.1: Search space and causal model. More relaxed causal model yields more inferred executions.

errors exposed by the given execution, it is called the *monitoring problem* (e.g. [67, 38, 102]). When the goal is to detect errors not only in the given execution, but also in other possible interleavings of the events of that execution, it is called the *prediction problem*.

Broadly speaking, existing predictive analysis techniques can be classified in two categories depending on how they infer new interleavings from an observed execution. Techniques that detect *must-violations* (i.e. report only real violations, no false positive) and techniques that detect *may-violation* (i.e. report potential violations).

Methods in the first category always predict feasible executions. Such methods often use under-approximated analysis, e.g. Lamport's happens-before causality relation and its extensions [29, 62, 83], the Maximal Causal Model (MCM) [85], SideTrack [106] the algorithms we present in Chapter 5 [41].

The idea is to identify causally equivalent execution without re-running the underlying program. This is achieved extracting causal partial orders from analyzing the events in the observed execution. Depending on how the causal partial order is defined the analysis can be more liberal or restrictive. Without additional information about the structure of the program under test, the least restrictive causal partial order that can be extract is the one in which each write event of a shared variable precedes all the corresponding subsequent read events and which is a total order on the events generated by each thread. This causality consider all the interaction among threads, it follows that the causal partial orders obtained allow a reduced number of linearizations and thus errors that can be exposed. In general, the larger the causality (as a binary relation) the fewer linearization it has, i.e., the more restrictive it is. In Figure 3.1 we report the search space (i.e. the set of possible thread interleavings or model states) of a concurrent program P. An observed execution of such program (green line), the causal model for the observed execution and the inferred executions. We report also an inferred buggy execution (red line).

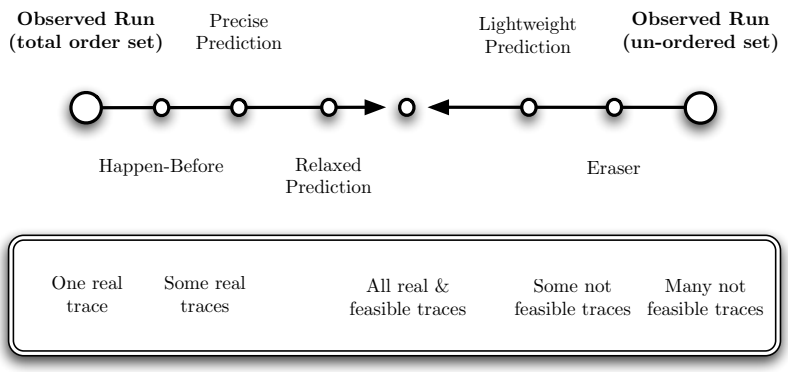


Figure 3.2: A reduced landscape of predictive analysis methods.

Methods in the second category often use over-approximated analysis, e.g. Eraser-style lock-set analysis [82], the algorithms we present in Chapter 4 [40, 91], and the UCG analysis based on universal causality graph [59]. Conceptually, methods in this group start by regarding the given observed run as an unordered set of events, meaning that any permutation of it is initially allowed, and then filtering out the obviously bogus ones using the semantics of the threads creation, synchronization primitives etc.

A simplified spectrum of predictive analysis methods is illustrated in Figure 3.2. Given an observed run, the left-most point represents the singleton set containing the observed run itself, whereas the right-most point represents the set of all possible permutations of it, regardless of whether the permutations are feasible or not. Consequently, the left-most point denotes the coarsest under-approximated predictive model, whereas the right-most point denotes the coarsest over-approximated predictive model.

Lamport [62] with the happens-before causality relationship originated the under-approximated line of research. During the years, the methods of this category have been able to cover more and more real bugs moving to the center of the horizontal line. The ground-breaking of the over-approximated methods is Eraser [82]. Over the years, even if improved the methods in this category may either miss many real bugs or generate many false alarms. In the middle of such lines of research there is the Concurrent Trace Program (CTP) model, proposed by Wang et al. [96, 97]. CTP represents the theoretically optimal point. However, the practical use of CTP as a predictive model depends on how efficient its error detection algorithm is. However, CPT models *computation*, and has too big an overhead to scale to the length and complexity of the runs that we handle in this work.

3.2 Our Approach

Our algorithms navigate the spectrum between under-approximated and over-approximated analysis realizing a bridge between precision and coverage. In the rest of the Section we give an overview of our approach.

3.2.1 Requirements

Given an observed execution σ we infer other executions σ' from σ. The semantic restrictions that σ' satisfies define its precision. In particular, given an inferred run σ', we always require that σ'_T is a prefix of σ_T, for every thread T. Additionally, lock-validity, creation-validity, data-validity, block-validity or a combination of them could be required. For example, lock-validity requires that σ' should respect the semantics of locks; data-validity requires that σ' should respect the semantics of reads, i.e. whenever a read of a value from a variable occurs, the last write to the same variable must have written the same value.

In Chapter 4 we achieve a lightweight prediction (fast and scalable) tracking only the control flow and synchronizations while ignoring the data flow, and the semantics of thread creation. On the other hand, in Chapter 5 we require also that the predicted run σ' respects data-validity, i.e. we use the Maximal Causality Model [85] to predict runs. MCM is the most precise prediction one can achieve at the level of shared communication.

Because we do not always require feasibility of the predicted runs, it follows that the rescheduling phase plays a crucial role in our prediction models. For the lightweight prediction algorithm, the rescheduling phase takes care of checking the feasibility of the predicting runs, getting rid of the false positives. However, the rescheduling phase is really important also for the precise prediction algorithm. It allows to target errors as null-pointer dereferences. Such errors can occur in a thread when dereferencing local variables. Without the rescheduling phase the prediction of of null-pointer dereferences would require handling local variables and the computation of threads.

3.2.2 Overview of Proposed Approach

Our framework, implemented in a tool called PENELOPE, works by taking a concurrent program P, a test input I, and a corresponding *set* of expected (correct) outputs $\mathcal{O}$ (we consider a set of outputs as concurrent programs can be non-deterministic). The input-output pair $(I, \mathcal{O})$ constitutes a test harness. We run the program P on the input I, and monitor (observe) an *abstracted* execution σ. σ records only the reads and writes to shared variables, synchronization events such as the acquisition and release of locks, thread creations, and barriers, and the begin and end of the sequential blocks of code if available (local computation, conditional checks, etc. are suppressed). In most cases, we will get an expected output $O \in \mathcal{O}$ at the end of the execution σ. We would like to know if there is an alternative schedule that leads to an execution σ' of the program which manifest some concurrency bug in P. In other

words, we would like to find an alternate execution σ' that generates an output O' that does not belong to the set of expected outputs $\mathcal{O}$, signaling the existence of a bug.

The predicted schedules, though feasible at the abstract level of observation, may not be actually feasible in the program P, and even if feasible, may not lead to actual errors the test harness attests to. However, our rescheduling phase takes care of this problem, getting rid of the false positives.
Our algorithms works in three phases.

- **Phase I: Monitoring.** In this phase we execute the program on the input from the test harness and observe a run σ. PENELOPE focuses its attention to read and write accesses to the (potentially) shared variables and acquisitions and releases of locks, thread creation, and barriers, and ignores all the other events.

- **Phase II: Prediction.** In phase II, PENELOPE constructs several runs σ', based on run σ, by carefully reordering the events in run σ so that each of the runs σ' could potentially expose a bug (e.g. null-pointer dereference, data-race, etc.).

- **Phase III: Rescheduling.** In the last phase, PENELOPE forces the program to execute σ' by weaving the threads using a single processor: if this is successful, then the generated output O' is checked to be in the set of expected outputs, $\mathcal{O}$; if this fails, a bug is reported. If PENELOPE fails to schedule a run σ', i.e. if σ' is not a feasible schedule of the program, or an error is not found, PENELOPE discards it and moves to another predicted execution.

3.3 Way to Find Bugs

PENELOPE implements both the lightweight and the precise prediction algorithms. We applied PENELOPE to a set of 18 concurrent programs exploring prediction of null-pointer exceptions, data-races, deadlocks and atomicity-violations to prove the predictive capabilities of our design. Specifically, PENELOPE has proved to be particularly effective and versatile, varying accuracy and scalability depending on the situations (more details in Chapter 7).

The algorithms we propose can be extended to the prediction of generic properties. In the remainder of this Chapter we present some of them.

3.3.1 Nondeterminism

Determinism is often a desired property in multi-threaded programs. A multi-threaded program is said to be deterministic if for a given input, different thread interleavings result in the same system configuration in the execution of the program.

It is important to consider when the system configuration is observed. If it is observed only at the end of the program execution, then individual read events may not need to read the same value across different interleavings. However, if the system configuration is continuously observed, then each read event must read the same value in all possible interleavings.

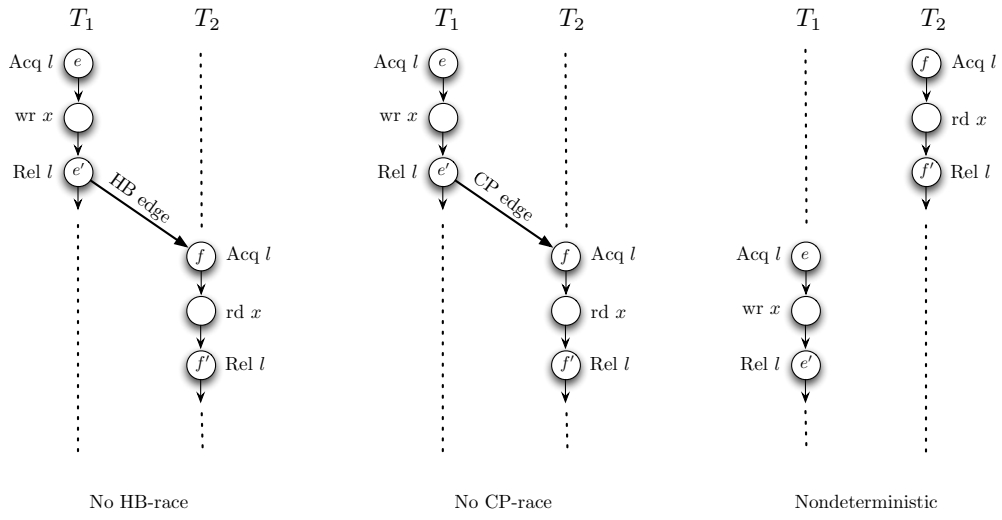


Figure 3.3: A deterministic program is race-free but the converse may not be true.

Notice that it exists a relation between deterministic and race-free programs. In particular a deterministic program is race-free but the converse may not be true (Figure 3.3).

In Figure 3.3, there is a pair of conflicting shared memory accesses, protected by the same lock l. Let events e and f indicate the acquisitions of lock l by T_1 and T_2 respectively. Similarly, let events e' and f' indicate the release of l by T_1 and T_2. If we consider the Happen Before (HB) analysis for lock operations, it follows that the events (e', f) are ordered by HB (edge in Figure) and hence there is no race. If we consider the causally precede (CP) analysis proposed by Smaragdakis et al. [89] there is no CP-race. Due to the presence of conflicting accesses (w and r) within the lock-scopes, e' causally precedes f introducing the CP edge.

However, another interleaving, where the lock-scopes swap order exists. That is, f' happens before e. This program is race-free but is nondeterministic because the read event (rd x) reads from a different write event in the interleaving.

The problem we want to solve can be formulated as follow:

Given a trace σ and a pair (w, r) in it, is there a write w' (with $w' \neq w$) such that it breaks (w, r) in another interleaving σ?

For a pair of events e and f and an interleaving σ, let $e \rightarrow f$ represents "e precedes f in the interleaving σ". For a given triplet (w, r, w') (all accessing the same shared variable), the read-couple is broken in an interleaving σ, when

any of the following orders is present in σ:

- $w' \rightarrow r \rightarrow w$
- $w \rightarrow w \rightarrow r$
- $w' \rightarrow r$ and w doest not occur in σ

This formulation allows us to easily accommodate this problem in our setting, in particular it could be considered as a variation of the null-pointer dereference prediction presented in Chapter 5.

3.3.2 Communication Deadlocks

In communication deadlocks some threads wait for messages that do not get sent because the sender threads are blocked or they have already sent the messages before receiving threads start to wait.

If in the observed execution of the concurrent program under test there is an event e in which T_i invokes the $wait()$ method on an shared variable x, followed later by an event f in which T_j invokes the $notify()$ method on the same variable x, we want to predict whether there is an alternate schedule that forces the event f to precede the event e, resulting in the lost of the $notify()$ call, and possibly in a deadlock configuration.

3.3.3 Generic API Violations

This idea of re-arranging methods/operations can be extended to more generic properties as violations of Java2SDK library (e.g. java.io). Specifically, properties that can be expressed as finite state automata and consequently encoded in a logical constraint solver [22, 63]. For example, the automaton in figure 3.4 represents the sequence of method calls executed on a java.io.OutputStream object that leads to an $IOexception$.

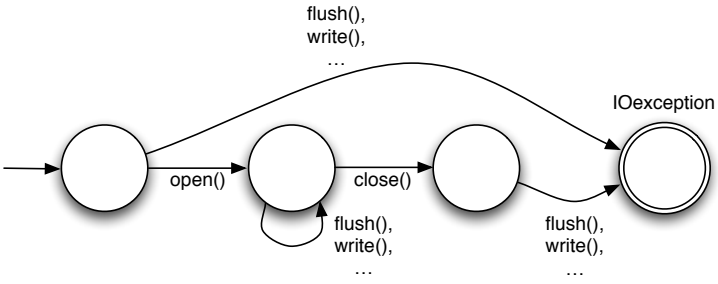


Figure 3.4: Sequence of method calls leading to an $IOexception$ on a $OutputStream$ object.

Chapter 4

Lightweight Prediction: Lock-sets Based Approach

The hypothesis of this work is that in order to test concurrent programs one can do interleaving selection more effectively by only exercising those interleavings that are symptomatic of common error patterns of interactions among threads. Each prediction algorithm starts from an observed run ρ of the program under test. The prediction algorithms are applied to ρ in order to obtains a new run ρ' that could potentially expose an error. Finally ρ' is passed to the re-scheduler module that force the program under test to run following exactly ρ' and detect if an error was found.

Notice that the events observed in the run change depending on the target of the prediction and on the precision we want to achieve.

In this Chapter, we focus on the common error patterns as *atomicity violations* and *deadlocks* and explore schedule selection algorithms that systematically choose and schedule those that violate atomicity or those that *potentially* deadlock.

A concurrent shared memory program, during its computation, does local computation, may read and write to shared entities (memory locations), dynamically create threads, and use synchronization primitives (such as locks). The framework we explore in this Chapter will observe only an abstraction of this computation that ignores local computation, ignores precise values read or written, but keeps track precisely of the read and write operations to shared memory locations as well as synchronization primitive usage. This abstract view of a computation, which we call an *execution*, will be the one that is algorithmically analyzed in order to predict and schedule alternate executions.

This setting as opposed to that presented in Chapter 5 implements a lightweight prediction. That is, the predicted model respects lock-validity, creation-validity, block-validity (if blocks are present) and the program-order of the original run; but it does not necessarily respect data-validity (Section 2.2). Such runs are certainly not guaranteed to be feasible in the original program— if the original program had threads that communicated through shared variables in a way that orchestrated the control flow, the predicted runs may simply not be feasible. However, in the absence of such communication, the predicted runs do respect the locking semantics and hence assure feasibility at that level of abstraction.

The rest of the Chapter is organized as follow. We start introducing the prediction model in Section 4.1 and then expose the prediction algorithms. Each prediction algorithm is composed by two phases; the first phase, presented in

Sections 4.2 and 4.3 for atomicity violations and deadlocks respectively, deals with the problem of checking whether there are alternate ways to schedule the events in the observed run to obtain buggy schedules. This is followed by the second phase, the schedule generation phase, which is common for atomicity violations and deadlocks. It actually synthesizes schedule based on the first phase and it is presented in Section 4.4.

4.1 Prediction Model

In order to develop theory around prediction algorithms we refresh some definitions presented in Section 2 and we introduce some new ones that will be used in the rest of the chapter. We assume an infinite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \ldots\}$ and an infinite set of global locks $\mathcal{L} = \{l_1, l_2, \ldots\}$, used in a nested fashion (i.e. threads release locks in the reverse order in which they were acquired).

Let ρ be a global execution, and $e = (T, i)$ be an event in $\{\rho_T\}_{T \in \mathcal{T}}$. Then we say that the j'th action ($1 \leq j \leq |\rho|$) in ρ *is the event* e (or, $Event(\rho[j]) = e = (T, i)$), if $\rho[j] = T{:}a$ (for some action a) and $\rho_T[1, i] = \rho[1, j]|_T$. In other words, the event $e = (T, i)$ appears at the position j in ρ in the particular interleaving of the threads that constitutes ρ. Reversely, for any event e in $\{\rho_T\}_{T \in \mathcal{T}}$, let $Occur(e, \rho)$ denote the (unique) j ($1 \leq j \leq |\rho|$) such that the j'th action in ρ is the event e, i.e $Event(\rho[j]) = e$. Therefore, we have $Event(\rho[Occur(e, \rho)]) = e$, and $Occur(Event(\rho[j])) = j$.

Finally, let $Tid(e, \rho)$ denote the thread $T \in \mathcal{T}$ executing the event e in ρ.

Definition 4.1.1 (Prediction model [91]) *Let ρ be a run over a set of threads $\mathcal{T}$ and locks $\mathcal{L}$. A run ρ' is* inferred *from ρ if (i) for each $T_i \in \mathcal{T}$, $\rho'|_{T_i}$ is a prefix of $\rho|_{T_i}$, (ii) ρ' is lock-valid, (iii) is creation-valid, (iv) is block-valid (whenever blocks are defined). We will refer to the set of executions inferred from ρ with $Infer(\rho)$.* ∎

We infer executions from ρ by projecting ρ to each thread to obtain local executions, and combining these local executions into a global execution ρ' in any interleaved fashion that respects the synchronization mechanism.

Notice that our prediction model is an *abstraction* of the problem of finding alternate executions. It is quite optimistic: it recombines executions in any manner that respects the locking constraints. Of course, these executions may not be valid in the original program (this could happen if the threads communicate using other mechanisms; for example, if a thread writes a particular value to a global variable based on which another thread chooses an execution path; in this case an execution that switches these events may not be valid). The choice of a simple prediction model is *deliberate*: while we propose a more accurate models in Chapter 5, we believe that having a simple prediction model can yield faster algorithms. Since we can anyway try to execute a predicted interleaving and check whether it is feasible, this will not contribute to the final false positives in a testing scenario.

```
public synchronized boolean addAll(Collection c) {
    modCount++;
    int numNew = c.size();
    // ....... possible interference ....
    ensureCapacityHelper(elementCount + numNew);
    Iterator e = c.iterator();
    for (int i=0; i<numNew; i++)
        elementData[elementCount++] = e.next();
    return numNew != 0;
}
```

Figure 4.1: Method `addAll` of concurrent Java class `Vector`.

4.2 Prediction of Atomicity-violating Schedules

We have chosen the class of executions that violate *atomicity* as a candidate for selecting schedules and validate our approach. A programmer writing a procedure often wants uninterfered access to certain shared data that will enable him/her to reason about the procedure locally. The programmer often puts together concurrency control mechanisms to ensure atomicity, often by taking locks on the data accessed. This is however extremely error-prone: errors occur if not all the required locks for the data are acquired, non-uniform ordering of locking can cause deadlocks, and naive ways of locking can inhibit concurrency, which forces programmers to invent intricate ways to achieve concurrency and correctness at the same time. Recent studies of concurrency errors [66] show that a majority of errors (69%) are atomicity violations. This motivates our choice in selecting executions that violate atomicity as the criterion for choosing interleavings to execute.

In this work we restrict to violation of minimal serializability, i.e. atomicity violation involving any two threads and any single variable according to the marked boundaries (see Section 2.4.1.1). The restriction to two threads and one variable is deliberate and pragmatically motivated— most atomicity errors occur due to two threads and one variable and our algorithms scale well in this case. The predicted schedules, however, may involve the scheduling of all threads, as these may be necessary to enable the violation in two threads.

4.2.1 Motivating Example

We use the following example to illustrate how our algorithm comes up with alternative runs that violate atomicity, which can find concurrency bugs in a program. Consider the implementation of the method `addAll` from the built-in Java library class `Vector` in Figure 4.1. The purpose of this method is to add all elements of the parameter collection (say another vector) to the end of the current vector. The method `size`, which returns the number of elements of the source, is a `synchronized` method. The problem in this program is that after safely retrieving the number elements that are to be copied, there can be a concurrent thread that modifies the source vector before the method finishes (for

example, a concurrent thread could remove all the elements from the source vector). Therefore, when the execution of the `addAll` method happens, the information about the number of the items that are being copied is stale, and an exception will be raised when it tries to access elements that are not there anymore.

Testing this program is unlikely to reveal these kinds of bugs in small methods such as `addAll`, as scenarios where the second thread gets interleaved in between the events of the first thread is unlikely. One has to actively make it happen, and that is exactly what we do.

This problematic scenario can be generalized using the following pattern of shared variable accesses:

1. T_1 reads the value of a shared variable (`c.size()` in the above example).
2. T_2 writes (modifies) the value of that same shared variable.
3. T_1 reads the value of the shared variable again (in `c.iterator()` in the above example).

We refer to such a pattern as a RWR pattern (Read-Write-Read). There are four more patterns that can capture other forms of undesired interference: WRW (Write-Read-Write), RWW (Read-Write-Write), WWR (Write-Write-Read), and WWW (Write-Write-Write). We focus on executions that contain one of these patterns as good candidates for finding hidden concurrency bugs.

4.2.2 Preliminaries

For the sake of readability let us refresh some notions introduced in Section 2 and here refined for the lightweight prediction of atomicity violations. The actions that a thread $T_i \in \mathcal{T}$ can perform on a set of shared variables SV and global locks $\mathcal{L}$ is defined as:

$$\Sigma_{T_i} = \{T_i{:}\,\textit{read}_x,\ T_i{:}\,\textit{write}_x |\ x \in SV\} \cup \{T_i{:}\,\textit{acquire}(l),\ T_i{:}\,\textit{release}(l) |\ l \in \mathcal{L}\} \cup \{T_i{:}\ tc\ T_j \mid T_j \in \mathcal{T}\}$$

We define $\Sigma = \bigcup_{T_i \in \mathcal{T}} \Sigma_{T_i}$ as the set of actions of all threads. Notice that we ignore local computation, ignore precise values read or written, but keep track precisely of the read and write operations to shared memory locations as well as synchronization primitive usage.

The *dependency* relation D, used to define the atomicity violation and to generate schedule (Section 2.4.1) is a symmetric relation defined over the events in Σ, which captures the dependency between (a) two events accessing the same entity, where one of them is a write, and (b) any two events of the same thread, i.e.,

$$\begin{aligned} D \quad = \quad & \{(T_1{:}a_1, T_2{:}a_2) \mid T_1 = T_2 \textit{ and } a_1, a_2 \in \Sigma \textit{ or } \exists x \in SV \textit{ such that } (a_1 = \text{read}(x) \textit{ and } a_2 = \text{write}(x)) \textit{ or} \\ & (a_1 = \text{write}(x) \textit{ and } a_2 = \text{read}(x)) \textit{ or } (a_1 = \text{write}(x) \textit{ and } a_2 = \text{write}(x))\} \end{aligned}$$

We can now define the precise problem we consider in this Subsection:

• **Problem of meta-analysis of executions for minimal serializability:**

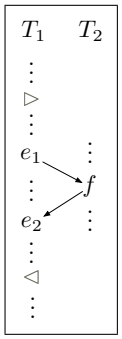
Given: A finite deadlock-free execution ρ over a set of threads, entities and locks.

Problem: Is there any execution $\rho' \in \mathit{Infer}(\rho)$ that is not minimally serializable?

4.2.3 Prediction Algorithm

In this Subsection, we present the basis for an algorithm that solves the meta-analysis problem for minimal serializability in time linear in the length of the given run. We will show how meta-analysis for minimal serializability can be reduced to the *global reachability problem* for two threads, which in turn can be compositionally and efficiently solved for nested-locking programs.

The results obtained in this Subsection will be used to formulate our algorithm in Section 4.2.4. The first observation is that only three events are relevant in finding a violation of minimal serializability; we need to observe two events e_1 and e_2 from a *single transaction* of a thread T_1 and an event f from another thread T_2 such that e_1 and f are dependent and e_2 and f are dependent. Moreover, and crucially, there should exist an execution in which f occurs after e_1, and e_2 occurs after f. The figure on the right describes this pattern, and the following lemma captures this property:



Lemma 4.2.1 *Let ρ be a global execution, and let $\{\rho_T\}_{T\in\mathcal{T}}$ be the set of local executions corresponding to it. $\mathit{Infer}(\rho)$ contains a minimally non-serializable run iff there exists two different threads T_1 and T_2, an entity $x \in \mathcal{X}$, and $\rho' \in \mathit{Infer}(\rho|_{\Sigma_{\{T_1,T_2\},\{x\}}})$ such that there are (read or write) events e_1, e_2, f of $\{\rho_T\}_{T\in\mathcal{T}}$ where*

- $\mathit{Occur}(e_1, \rho') < \mathit{Occur}(f, \rho') < \mathit{Occur}(e_2, \rho')$
- *e_1 and e_2 are events of thread T_1, and f is an event of thread T_2*
- *e_1 and e_2 belong to the same transaction,*
- *$e_1\ D\ f\ D\ e_2$.*

While we can find candidate events e_1 and e_2 from thread T_1 and a candidate event f from T_2 by *individually* examining the local runs of T_1 and T_2, the main problem is in ensuring the condition that we can find an inferred run where e_1 occurs before f and f occurs before e_2. This is hard as the threads synchronize using locks which needs to be respected by the inferred run. In fact, for threads communicating using locking, Farzan et al. [38] show that it is highly unlikely to avoid considering the two thread runs in tandem, which involves $O(n^2)$ time.

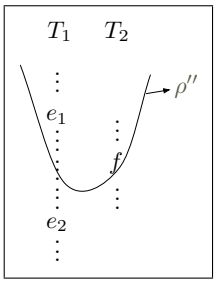
Let us first show the following lemma that reduces checking whether three events e_1, f, and e_2 are executable in that order, to global reachability of two threads.

Lemma 4.2.2 *Let ρ be a deadlock-free execution, and let T_1, T_2 be two threads with $T_1 \neq T_2$. Let e_1, e_2, f be (read or write) events in $\{\rho_T\}_{T \in \mathcal{T}}$ such that $e_1 = (T_1, i_1)$ and $e_2 = (T_1, i_2)$ are events of thread T_1 with $i_1 < i_2$, and f is an event of thread T_2. Then, there is an execution $\rho' \in Infer(\rho)$ such that $Occur(e_1, \rho') < Occur(f, \rho') < Occur(e_2, \rho')$*

if, and only if,

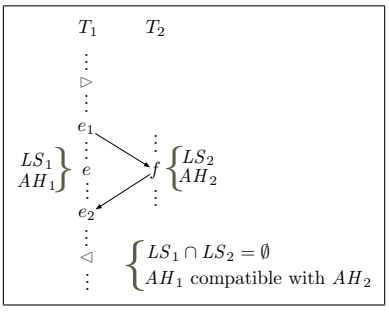
there is an execution $\rho'' \in Infer(\rho)$ such that

- *e_1 occurs in ρ'' and e_2 does not occur in ρ'', and*
- *f occurs in ρ'', and in fact f is the last event of T_2 that occurs in ρ''.*



Intuitively, the above lemma says the following: fix an execution ρ, and three events e_1, e_2, f in it such that events e_1 and e_2 belong to the same transaction (and thread) and event f belongs to a different thread. Then, we can find a run inferred from ρ that executes event e_1 followed by event f followed by event e_2, if, and only if, we can find an (incomplete) inferred run that executes events events e_1 of thread T_1 (and possibly later events), but does not execute e_2, and executes precisely up to event f in thread T_2. This is depicted in the figure on the right.

The above lemma is useful as it reduces finding a series of three events to the simpler global reachability question of a set of pairs of positions in the two threads (see Section 2.3.1).



Consider a typical execution that we are looking for, one where e_1 occurs first, then f, and then e_2, where e_1 and e_2 occur in one transaction block while f occurs in another thread, and the three events form one of the patterns violating atomicity. The basic argument is that such an abstract lock-valid execution is possible if and only if there is an intermediate event e between e_1 and e_2 (e can be e_1 but not e_2) such that there is an execution that reaches exactly up to executing e and f in the two threads. To see why, notice that we can roll back f and play it last (as it is an access to a variable), and hence we can execute e_1 followed by f, and let the run proceed to execute e_2 eventually.

It now turns out that checking whether there is a lock-respecting run that reaches simultaneously e and f is easily solvable, simply by examining the lock-sets and acquisition histories at these two points, which gives a necessary and sufficient condition for it: the lock-sets at e and f must be disjoint and the acquisition histories at e and f must be compatible [58]. Let us call a *c*ut-point the pair of events in the execution (e, f) such that there is an alternate schedule

that can reach exactly up to e and f simultaneously. Any schedule that reach exactly up to e and f simultaneously gives an atomicity violation. The above arguments give a characterization of all cut-points that lead to atomicity violations. We summarize this in the following:

Theorem 4.2.3 *Let ρ be a deadlock-free global execution. A minimally non-serializable execution can be inferred from ρ iff there exists two different threads T_1 and T_2 and an entity $x \in SV$, and there are events $e_1 = (T_1, i), e_2 = (T_1, i'), f = (T_2, j)$ of $\{\rho_T\}_{T \in \mathcal{T}}$ such that*

- *e_1 and e_2 belong to the same transaction,*
- *$e_1 \; D \; f \; D \; e_2$,*
- *There is an event $e = (T_1, i'')$ of $\{\rho_T\}_{T \in \mathcal{T}}$ such that $i \leq i'' < j$ and the acquisition histories of $\rho_{T_1}[1, i'']$ and $\rho_{T_2}[1, j]$ are compatible.*

On the assumption of deadlock freedom: Deadlock freedom ensures that the partial run (containing e_1 and f) can be completed to a full run where e_2 occurs. We can remove the assumption of deadlock freedom and build a more sophisticated algorithm (using lock-set and *b*ackward acquisition histories) that ensure that e_2 is also executed; however this complicates the algorithm considerably, and we have chosen not to implement it for practically reasons. In the next Subsection we propose an algorithm to predict deadlocks, making the deadlock freedom assumption reasonable. We will actually see that under the assumption of deadlock freedom, checking compatibility of acquisition histories is redundant, as it suffices to check if lock-sets are disjoint. However, we check compatibility of acquisition histories in order not to rely on the assumption of deadlock freedom to generate the events e_1 and f.

4.2.4 Cut-points Generation Algorithm

Given a set of local executions $\{\rho_T\}_{T \in \mathcal{T}}$ with nested locking, Theorem 4.2.3 allows us to build an efficient algorithm to predict an interleaving of them that violates minimal serializability.

The aim is to find three events e_1, e_2, and f, where e_1 and e_2 occur in the same transaction in a thread T_1, f occurs in a different thread T_2, with $e_1 D f D e_2$, and further, find an event e between e_1 and e_2 (e is allowed to be e_1 but not to be e_2) such that the lock-sets of e and f are disjoint, and their acquisition histories are compatible.

The algorithm is divided into two phases. In the first phase, it gathers the lock-set and acquisition histories of all possible witnesses e and all possible witnesses f; this is done by examining the events of each thread *individually*. In the second phase, we test the compatibility of the lock-sets and acquisition histories of every pair of witnesses e and f in different threads.

Let us fix an entity $x \in SV$. We divide our work into finding two patterns: one where e_1 and e_2 are writes to x and f is a read of x, and the other where e_1 and e_2 are accesses (read/write) to x and f is a write to x. This clearly covers all

cases of minimal serializability violations— the former covers violations $e_1 - f - e_2$ of the form $Write$–$Read$–$Write$, while the latter covers those of the form $Read$–$Write$–$Read$, $Read$–$Write$–$Write$, $Write$–$Write$–$Read$ and $Write$–$Write$–$Write$.

In the first phase, for each thread T and each entity x, the algorithm gathers witnesses in four lists: $R[T, x]$, $W[T, x]$, $WW[T, x]$ and $AA[T, x]$. Intuitively, the sets $R[T, x]$ and $W[T, x]$ gather witnesses of events of thread T that read and write to x, respectively, for each lock-set and acquisition history pair, in order to witness the event f in our pattern.

The set $WW[T, x]$ gathers all witnesses e that are sandwiched between two write-events to x that occur in the same transaction of thread T, keeping only one representative witness for each lock-set and acquisition history pair. Similarly $AA[T, x]$ gathers witnesses e sandwiched between any two accesses of thread T to x that occur in the same transaction.

The algorithm gathers the witnesses by processing the execution in a single pass. It continually updates the lock-set and acquisition history, adding events to the various witness sets, making sure that no set has multiple events with the same lock-set and acquisition history. Note that the computation of $WW[T, x]$ and $AA[T, x]$ sets need care due to the fact that events e recorded must be validated by a later occurrence of the relevant event e_2.

Note that in the first phase it is considered every event at most once, and hence it runs in time linear in the length of the execution.

In the second phase, the algorithm checks whether there are pairs of compatible witnesses that were collected in the first phase. More precisely, we check whether, for any entity x, and for any pair of threads T_1 and T_2, there is an event $e \in WW[T_1, x]$ and an event $f \in R[T_2, x]$ that have disjoint lock-sets and compatible acquisition histories. Similarly, we also check whether there is an event $e \in AA[T_1, x]$ and an event $f \in W[T_2, x]$ that have disjoint lock-sets and compatible acquisition histories. The existence of any such pair of events would mean (by Theorem 4.2.3) that there is a minimal serializability violation.

For example, the algorithm runs the procedure in Figure 4.2 for finding the violations using the R and WW sets (the procedure using the W and AA sets is similar):

```
1  for each entity x ∈ V do
2    for each T1, T2 in 𝒯 such that T1 ≠ T2 do
3      for each (e, LS, AH) in WW[T1, X] do
4        for each (f, LS', AH') in R[T2, x] do
5          if (LS ∩ LS' = ∅  ∧  AH and AH' are compatible) then
6            Report minimal serializability violation found;
7  end
```

Figure 4.2: Second phase for R-WW patterns.

Note that the second phase runs in time $O(t^2.v.ah^2)$ where t is the number of threads, v is the number of entities accessed, and ah is the total number of disjoint acquisition histories of events in the thread. Note that this is *independent* of the length of the execution (the first phase summarized the events in the execution, and the second phase does not consider the run again).

The quadratic dependence on the number of threads is understandable as we consider serializability violation between all pairs of threads. The linear dependence on x is very important for scalability as the number of entities accessed can be very large on typical runs. The number of different acquisition histories, in theory, can be large ($O(2^{l^2})$, where the execution uses l locks)— however, in practice, there tend to be very few distinct acquisition histories that get manifested, and hence is not a bottleneck (see Chapter 7 for details).

The algorithm presented in this Subsection records violations only in terms of the two witnesses e and f, we can actually recover a precise execution that shows the atomicity violation. This run can be obtained using the lock-sets and acquisition histories of e and f. It may in fact involve several context switches ($O(l)$ of them, if there are l locks) to execute the atomicity violation and it is presented in Section 4.4.

4.3 Prediction of Deadlocking Schedules

A common cause for unreactiveness in concurrent programs are deadlocked configurations. A set of threads is deadlocked if each thread deadlocked is waiting for a resource, like a lock, held by another thread in the set, which forms a cycle of lock requests. Deadlocks often occur under subtle interleaving patterns that the programmer has not taken into consideration. In this Subsection, we focus on *prediction techniques for discovering deadlocks*— we observe an arbitrary execution of a concurrent program and from it predict alternate interleavings that can deadlock. We show that if a concurrent program adopts *nested locking* policies, the problem of predicting potential deadlocks involving any number of threads is efficiently solvable without exploring all interleavings.

The prediction algorithm, which reasons at an abstract level in order to efficiently and accurately predict deadlocking schedules; it is based on *lock-sets* and *acquisition histories* (see Section 2.3), which only ensure that the predicted run respects *lock* acquisitions and releases in the run.

4.3.1 Motivating Example

It is very common to incur a deadlock when programs misuse APIs offered by third-party libraries [57, 101]. Even if the program does not contain logic bugs per se, the interaction of methods defined in synchronized classes may still result in deadlocks. This is a general problem for all synchronized Collection classes in the `JDK`, including the `Vector` library. Since `Vector` is a synchronized class, programmers could easily assume that concurrent accesses

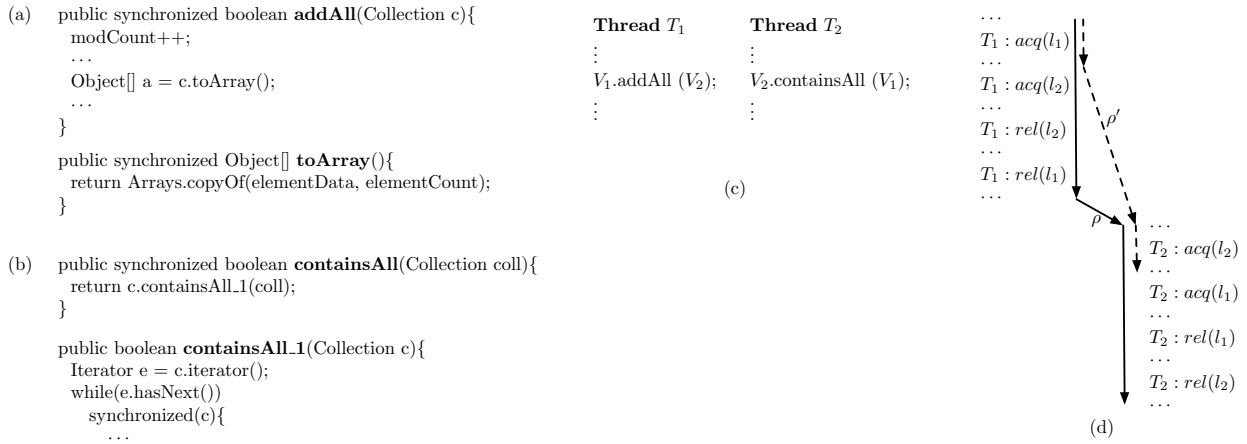


Figure 4.3: (a) – Simplified code for the methods $addAll$ and $toArray$ of the $Vector$ library of Java 1.4. (b) – Simplified code for the methods $ContainsAll$ and $ContainsAll\_1$ of the $Collections$ library of Java 1.4. (c) – Code executed concurrently by the threads T_1 and T_2. (d) – Observed execution ρ of the program under test (dotted arrows indicate the predicted run ρ' generated from ρ).

to vectors are not a concern. However, potential deadlocks could still be present and hidden from the calling application. Ideally, APIs should be documented carefully; unfortunately, this is not always the case in practice. Moreover, developers cannot think of every possible way in which their APIs will be used.

In Figure 4.3(a), we report a simplified version of the methods $addAll$ and $toArray$ as defined in the `JDK` library. $addAll$ appends all of the elements in the specified collection c to the end of $this$ Vector. Internally it calls the method $toArray$, which returns an array containing all the elements in $this$ Vector in the correct order. As the $addAll$ needs to be multi-thread safe, it follows that locks for the vector being added to and the parameter need to be acquired. Specifically, the method acquires the lock associated with the vector being added to first, then it acquires the lock associated with the parameter. Similarly, in Figure 4.3(b), we report a simplified version of the methods $containsAll$ and $containsAll\_1$ ($containsAll$ defined in AbstractCollection.java) as defined in the `JDK` library. $containsAll$ acquires the lock associated with $this$ vector first, then from inside the method $containsAll\_1$ it acquires the lock associated with the specified collection c.

In Figure 4.3(c), we report the code executed concurrently by the two threads T_1 and T_2 that use two vectors, V_1 and V_2. T_1 wants to add all elements of V_2 to V_1, calling the method V_1.addAll(V_2), while T_2 concurrently invokes the method V_2.containsAll(V_1), in order to check if the vector V_2 contains all the elements of V_1.

Finally, in Figure 4.3(d), we report a possible observed run ρ of the program under test (ignore the dotted arrows). We are assuming that the code of T_1 is entirely executed followed by the code of T_2 (this execution does not deadlock). Our prediction algorithm observes the synchronization events (such as locks acquire/release) but suppresses the

semantics of computations entirely and does not observe them. They have been replaced by "..." in the figure as they play no role in our analysis.

Given the observed run ρ, we ask whether there exists an alternative run ρ' in which a deadlock potentially occurs. Our prediction algorithm will predict a run ρ' in which the acquisition of the lock associated with V_1 (let us say l_1) by T_1 is followed by the acquisition of the lock associated with V_2 (let us say l_2) done by T_2 (illustrated by the dotted arrows in the figure).

4.3.2 Preliminaries

Let us re-define the set of actions of all threads $\Sigma = \bigcup_{T_i \in \mathcal{T}} \Sigma_{T_i}$. The actions that a thread $T_i \in \mathcal{T}$ can perform on a set of shared variables SV and global locks $\mathcal{L}$ is defined as:

$$\Sigma_{T_i} = \{T_i{:}\,\textit{acquire}(l),\ T_i{:}\,\textit{release}(l) |\ l \in \mathcal{L}\} \cup \{T_i{:}\ tc\ T_j \mid T_j \in \mathcal{T}\}$$

In this prediction model we work at the level of abstraction. We ignore local computation, ignore the read and write operations to shared memory locations but we keep synchronization primitive usage.

• **Problem of meta-analysis of executions for deadlock:**

Given: A finite execution ρ over a set of threads, entities and locks.

Problem: Is there any execution $\rho' \in \mathit{Infer}(\rho)$ that is deadlocking?

Our prediction model is an *abstraction* of the problem of finding alternate executions that are deadlocking in the concrete program. Not all the executions in $\mathit{Infer}(\rho)$ may be valid/feasible in the original program (for the same reasons stated in the previous Section). In this sense we talk about *potential deadlocks*. A precise precisely predictable model can be obtained adding to Def. 4.1.1 the *data-validity* constraints (see 5). However, our rescheduling phase takes care of this problem, getting rid of the false positives.

4.3.2.1 Relation Between Co-reachability and Deadlock

Lemma 2.3.1 argues that global reachability of two threads communicating via nested locks is effectively and compositionally solvable by extracting locking information from the two threads in terms of acquisition histories. In particular, it states that there is an execution that ends with event e_1 in one thread and event e_2 in the other thread, if, and only if, the acquisition histories at e_1 and e_2 are compatible and the lock-sets held are disjoint.

This pairwise reachability result is the base of our approach and the following Theorem is a direct consequence of it.

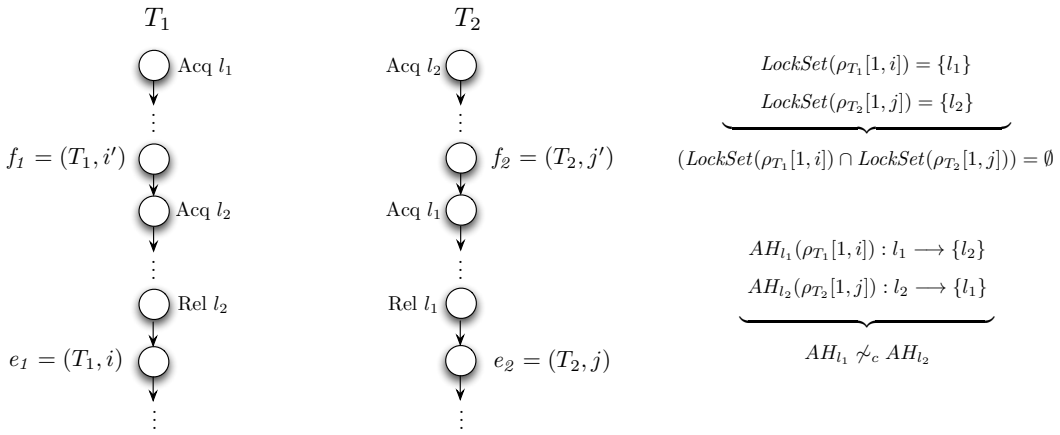


Figure 4.4: Lock-sets and acquisition histories associated with a deadlocking configuration of threads T_1 and T_2.

Theorem 4.3.1 *There is a potential deadlocking run $\rho' \in Infer(\rho)$ involving two threads*

if, and only if,

$\exists\, e_1 = (T_1, i)$, $e_2 = (T_2, j)$ *s.t.* $LockSet(\rho_{T_1}[1,i]) \cap LockSet(\rho_{T_2}, [1,j]) = \emptyset$ *and the acquisition histories of* $\rho_{T_1}[1,i]$ *and* $\rho_{T_2}[1,j]$ *are not compatible.*

Proof: One side of the implication follows from the Lemma 2.3.1. If there is a deadlock it means that we reached an event $f_1 = (T_1, i')$ and an event $f_2 = (T_2, j')$ in the local executions in which the two threads are not allowed to make further operations (Figure 4.4). From the Lemma 2.3.1, it follows that lock-sets at f_1 and f_2 are disjoint and the acquisition histories of $\rho_{T_1}[1, i']$ and $\rho_{T_2}[1, j']$ are compatible. Moreover, because the threads T_1 and T_2 are blocked, the operation that they are trying to do is an acquire of some lock (the release is not a blocking operation).

In particular, T_1 and T_2 are trying to acquire different locks, because if they were trying to acquire the same lock at least one of the threads would have been able to move. The fact that T_1 (resp. T_2) can not make an acquire implies that it is requiring a lock, l_2 (resp. l_1) owned by T_2 (resp. T_1). It follows that at $\rho_{T_1}[1, i' + 1]$ and $\rho_{T_2}[1, j' + 1]$, the lock-sets are not disjoint.

From the nested nature of the locking policies, it follows that there exists a point in ρ_{T_1} in which l_2 is released, that has the same lock-set of f_1, let us say e_1. Similarly, there exists a point in ρ_{T_2} in which l_1 is released, that has the same lock-set of f_2, let us say e_2. It follows that at e_1 and e_2 the lock-sets are disjoint (they were disjoint also in f_1 and f_2). Because T_1 held l_1 while acquired l_2 and T_2 held l_2 while acquired l_1 then l_1 and l_2 are such that the acquisition histories of $\rho_{T_1}[1, i]$ and $\rho_{T_2}[1, j]$ are not compatible, that completes this side of the proof.

It remains to prove that when $\exists e_1, e_2$ satisfying the hypothesis then there exist an event, f_1, executed by T_1 with $Occur(f_1, \rho) < Occur(e_1, \rho)$, and an event, f_2, executed by T_2 with $Occur(f_2, \rho) < Occur(e_2, \rho)$, such that T_1 and T_2 are deadlocked.

Let us pick the e_1 e_2 such that $(Occur(e_1, \rho) + Occur(e_2, \rho))$ is minimal, moreover for the sake of exposition we can assume that there is a unique pair of locks (l_1, l_2) such that the acquisition histories of $\rho_{T_1}[1, i]$ and $\rho_{T_2}[1, j]$ are not compatible. We need to prove that f_1 and f_2 respectively in T_1 and T_2 are co-reachable and deadlocking.

From the assumption it follows that in ρ_{T_1} and in ρ_{T_2}, before that the events e_1 and e_2 are respectively executed, the locks l_1 and l_2 are acquired in reverse order by T_1 and T_2. We can assume that the execution orders are those depicted in Figure 4.4.

We pick as f_1 the event right before the acquisition of l_2 in ρ_{T_1} and the event right before the acquisition of l_1 in ρ_{T_2} as f_2.

f_1 and f_2 are deadlocking by definition. In order to prove that they are co-reachable, from Lemma 2.3.1, we need to prove:

1. $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset$.
2. acquisition histories at $\rho_{T_1}[1, i']$ and $\rho_{T_2}[1, j']$ are compatible.

Due to the nested nature of the locking policies, the event f_1 occurs after the acquisition by T_1 (resp. T_2) of l_1 (resp. l_2), belonging to $LockSet(\rho_{T_1}[1, i])$ (resp. $LockSet(\rho_{T_2}[1, j])$). It follows $LockSet(\rho_{T_1}[1, i]) \subseteq LockSet(\rho_{T_1}[1, i'])$ and $LockSet(\rho_{T_2}[1, j]) \subseteq LockSet(\rho_{T_2}[1, j'])$.

By contradiction, let us assume that $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) \neq \emptyset$. That is, it exists a lock l_3 such that $l_3 \in LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j'])$. But from the inclusions stated above it follows that $l_3 \in LockSet(\rho_{T_1}[1, i]) \cap LockSet(\rho_{T_2}[1, j])$ that contradicts the hypothesis.

We can conclude that the lock-sets are disjoint when T_1 is at $\rho_{T_1}[1, i']$ and T_2 is at $\rho_{T_2}[1, j']$. It remains to prove the point 2. Let us assume by contradiction that exist l_1 and l_2 such that the acquisition histories at $\rho_{T_1}[1, i']$ and $\rho_{T_2}[1, j']$ are not compatible. We found two events satisfying the hypothesis, moreover $Occur(f_1, \rho) < Occur(e_1, \rho)$ and $Occur(f_2, \rho) < Occur(e_2, \rho)$ it follows that $(Occur(e_1, \rho) + Occur(e_2, \rho))$ was not minimal, contradicting the assumption. ∎

4.3.2.2 The Importance of Acquisition Histories

Potential deadlocks could be detected using a multitude of approaches. The question we want to address in this Section is: *"Why use acquisition histories?"*. The ideal prediction algorithm would predict potential deadlocks that are feasible at least with respect to the synchronization mechanisms.

The majority of the approaches that have been proposed in literature are based on cycles detection in *lock order graphs* [21, 24]. In a lock order graph, a node represents a lock. A directed edge from node l_1 to node l_2 labeled T represents that, during the execution, the thread T acquires the lock l_2 while holding the lock l_1. For the rest of this

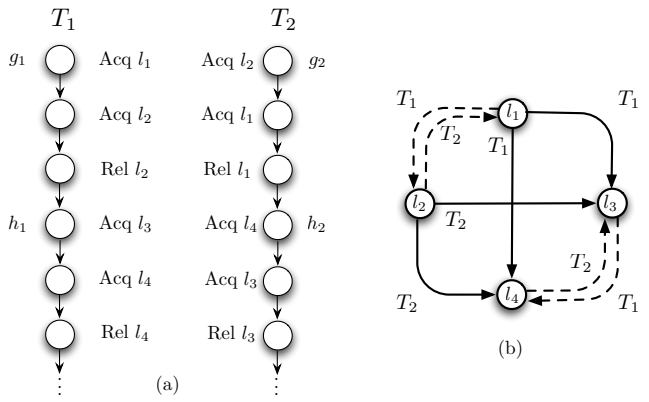


Figure 4.5: (a) – A problematic scenario for the *next lock required* and the *cycle detection* approaches. (b) – Lock order graph associated with the execution on the left (dotted lines are detected cycles).

Section we consider deadlocks involving only two threads for the sake of exposition.

Let us consider a program in which two threads T_1 and T_2 run concurrently and they use four locks l_1, l_2, l_3 and l_4. Given an execution ρ of such program, in Figure 4.5(a) we report the local executions ρ_{T_1} and ρ_{T_2} and the lock order graph associated with it (b). The classic technique based on cycles detection [21, 24, 27, 56, 68], first constructs a lock order graph. Then it detects whether there are any cycles on the graph (in Figure 4.5(b) dotted lines are detected cycles). Finally, it tries to trigger the potential deadlock using active scheduling strategies.

Another simple algorithm to detect potential deadlocks keeps the set of locks (for each event) held by the thread when an event is executed (similar to what we do), and also keeps the information about *next lock required*. A potential deadlock is found when given two events e_1 and e_2 (executed by two distinct threads), they have disjoint lock-sets and the next lock required at e_1 (resp. e_2) is in the lock-set associated with e_2 (resp. e_1).

Consider the local executions in Figure 4.5(a). The next lock required approach will report a potential deadlock at (g_1, g_2). Moreover, it will report a potential deadlock at (h_1, h_2). Notice that these two potential deadlocks are also reported by the lock order graph-based approach. Unfortunately, (h_1, h_2) is not a reachable configuration. Magic-Fuzzer [27] is the most recent technique based on lock order graph. With respect to its competitors, this approach is more efficient because the size of the graph built is one order of magnitude smaller then similar approaches. It iteratively prunes lock dependencies that each has no incoming or outgoing edge. However, even this approach will wrongly report the two potential deadlocks.

The algorithm we propose is precise and lightweight. Precise because it will report a potential deadlock configuration only when it is feasible (at least respect to the synchronization mechanisms), potentially saving significant aggravation to the user if a manual inspection of the potential deadlocks is required (or if a re-execution phase is provided). It is lightweight because any deadlock prediction algorithm would keep track at least of lock-sets. Once, that a potential

deadlock is found a mechanism to expose it is still needed. This is what acquisition histories help us to do.

4.3.2.3 Concise Deadlock Prediction

A set of threads is deadlocked if each thread in the set requests a lock, held by another thread in the set, forming a cycle of lock requests. A cycle with n components is a sequence, but there are n total permutations of the components to represent the same cycle. Detecting one permutation suffices to represent the cycle. In a cycle, each thread can occur only once [27]. We can then use a thread-driven approach to consider only one permutation in place of the whole set of permutations representing the same cycle.

Additionally, the algorithm we propose predicts deadlocks which have the minimal number of threads involved. Our algorithm is run repeatedly, it starts looking for deadlocks involving only two threads. If no deadlocks are found the algorithm proceeds looking for deadlocks involving three threads and so on until a potential deadlock is found or all the possible combinations of threads have been considered. If the potential deadlock predicted is real, this feature will be very helpful for the developer through the debugging process. More details will be discussed in the next Section.

4.3.3 Prediction Algorithms

Given an execution ρ with nested locking, we would like to *infer* other executions ρ', containing a deadlock, from ρ. Theorem 4.3.1 allows us to engineer an efficient algorithm to predict deadlocking executions. Our algorithm is based on the pairwise reachability, which is solvable compositionally by computing lock-sets and acquisition histories for each thread. In this Subsection we will consider first the prediction of deadlocks involving only two threads and then the more general case involving any number of threads.

4.3.3.1 Deadlocks Prediction: 2-threads

The aim of the algorithm is to find two deadlocking events, executed by two distinct threads, given an observed execution ρ. Notice that we are looking for potential deadlocks at this point. These deadlocks may not be feasible in the original program (this could happen if the threads communicate using other mechanisms; for example, if a thread writes a particular value to a global variable which another thread uses to choose an execution path).

The algorithm is divided into three phases. In the first phase, it gathers the lock-sets and acquisition histories by examining the events of each thread *individually*. In the second phase, it tests the compatibility of the lock-sets and acquisition histories of every pair of witnesses e_1 and e_2 in different threads, collected in the first phase. In the third phase from such e_1 and e_2 it rolls back to two co-reachable events, f_1 in T_1 and f_2 in T_2 respectively, such that the two threads are deadlocked.

```
1  for each x, y ∈ AH^1_ρ do
2    if Tid(ev(x)) > Tid(ev(y)) then
3      if ls(x) ∩ ls(y) = ∅  ∧  ah(x) ⊀_c ah(y) then
4        pass the pair (x, y) to the third phase;
5  end
```

Figure 4.6: Phase II.

Phase I. In the first phase, the algorithm gathers witnesses for each thread T. The algorithm gathers the witnesses by processing the local executions ρ_T in a single pass. It continuously updates the lock-set and acquisition history, adding events to the set of witnesses, making sure that there are no events with the same lock-set and acquisition history.
The set of witnesses, indicated with AH^1_ρ, is a set of 3-tuples $\{((T, i), \mathit{LockSet}(\rho_T[1, i]), AH(\rho_T[1, i])) \mid T \in \mathcal{T}, 1 \leq i \leq |\rho_T|\}$. We use corresponding projection functions $ev(x)$, $ls(x)$ and $ah(x)$ to extract the components from $x \in AH^1_\rho$.
Note that phase I considers every event at most once, in one pass, in a streaming fashion, and hence runs in time linear in the length of the execution.

Phase II. In the second phase, the algorithm checks whether there are pairs of not compatible witnesses collected in the first phase. More precisely, it checks whether, for any pair of threads T_1 and T_2, there is an event e_1 executed by T_1 and an event e_2 executed by T_2 in AH^1_ρ that have disjoint lock-sets and not compatible acquisition histories (also indicated with $\not\sim_c$). The existence of any such pair of events would mean (by Theorem 4.3.1) that there is a potential deadlock configuration involving the threads T_1 and T_2.
The algorithm runs the procedure in Figure 4.6 for finding deadlocks:

Notice that the condition $>$ (in place of $\neq$) on line 2 avoids reporting redundant deadlocks, as mentioned in Section 4.3.2.3. This reduction is sound because the lock-sets intersection and the acquisition histories compatibility are commutative operations.

Phase III. In the third phase, the algorithm retrieves two deadlocking events f_1, f_2 from a pair of events (e_1, e_2) generated in the second phase. Acquisition histories and the lock-sets of e_1 and e_2 are used to backtrack until an appropriate deadlocking configuration (f_1, f_2) is found. The algorithm stops the backtracking process when the two events f_1 and f_2 have disjoint lock-set and compatible acquisition histories.

In particular, given a pair of events (e_1, e_2) indicating the presence of a deadlock (i.e. e_1 and e_2 have disjoint lock-set and not compatible acquisition histories) we want to retrieve a pair of deadlocking events (f_1, f_2). Let us call a *c*ut-point the pair of events in the execution (f_1, f_2) such that there is an alternate schedule that can reach exactly up to f_1 and f_2 simultaneously. Any schedule that reach exactly up to f_1 and f_2 simultaneously gives a deadlock. The algorithm to retrieve the cut-point runs the procedure in Figure 4.7. Essentially T_1 and T_2 are backtracked at the events were the problematic locks were acquired (lines 2-3). Let us assume that T_1 holds l_1 when acquires l_2 at $f_1 = (T_1, i')$.

Parameters $e_1 = (T_1, i)$,$e_2 = (T_2, j)$
1 **for** *each* l_1, l_2 s.t. $l_2 \in AH_1(l_1)$ and $l_1 \in AH_2(l_2)$ with $AH_1 \in AH(\rho_{T_1}[1, i])$ and $AH_2 \in AH(\rho_{T_2}[1, j])$ **do**
2 $f_1 = (T_1, i')$ in ρ_{T_1} with $i' < i$ and the action performed is $T_1 : acquire(l_2)$
3 $f_2 = (T_2, j')$ in ρ_{T_2} with $j' < j$ and the action performed is $T_2 : acquire(l_1)$
4 **if** $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) \neq \emptyset \ \vee \ AH(\rho_{T_1}[1, i']) \not\sim_c AH(\rho_{T_2}[1, j'])$ **then**
5 return null;
6 **else** $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset \ \wedge \ AH(\rho_{T_1}[1, i']) \sim_c AH(\rho_{T_2}[1, j'])$ **then**
7 return the cut-point (f_1, f_2);
8 **end**

Figure 4.7: Phase III.

Similarly, T_2 holds l_2 when acquires l_1 at $f_2 = (T_2, j')$.

If $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset \ \wedge \ AH(\rho_{T_1}[1, i']) \sim_c AH(\rho_{T_2}[1, j'])$, we have found two co-reachable deadlocking point (line 6) and we return the pair (f_1, f_2) (line 7).

If the f_1 and f_2 are not co-reachable (line 4) then we have two possible cases: (1) $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) \neq \emptyset$ or (2) $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset$ and $AH(\rho_{T_1}[1, i']) \not\sim_c AH(\rho_{T_2}[1, j'])$. In both cases there is another deadlock involved and consequently in AH^1_ρ there are two events x' and y' related to it, and that will be considered in Phase II. Then we can interrupt the retrieving process (line 5).

4.3.3.2 Deadlocks Prediction: n-threads

Unfortunately, the lock-set/acquisition history analysis we considered is unable to catch potential deadlocks involving more than two threads. We illustrate the n-threads deadlock prediction using the classical *Dining Philosophers Problem*. Four philosophers, identifiable by a unique id $i \in \{1, 2, 3, 4\}$, sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. A philosopher i can only eat spaghetti when she has both left and right forks (resp. Fi and $F((i + 1)mod4)$). When the philosophers get hungry is not deterministic. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. When a philosopher i is hungry she tries to acquire her left fork first $F(i)$, and once she obtained that she tries to acquire the right fork $F((i + 1)mod4)$. After she finishes eating, she needs to put down both the forks so they become available to others. In particular, she puts the right fork down first and then the left fork. The code for the program is adapted from [28] and it is reported in Figure 4.8. In Figure 4.9 we report the local executions for the threads involved in the Dining Philosophers program. All the acquisition histories associated with the events of T_1, T_2, T_3 and T_4 are pairwise compatible, and the lock-sets are pairwise disjoint. It follows that no potential deadlocks are found in this case. The problem is that the elements in AH^1_ρ are built from single thread information, therefore in order to increase the power of the lock-set/acquisition history analysis, we need a set whose elements synthesize the information of multiple threads. We present in the following the algorithm for the detection of potential deadlocks involving multiple

```
class Philo {
  public static void main(String[] args) {
    Fork F1 = new Fork();
    Fork F2 = new Fork();
    Fork F3 = new Fork();
    Fork F4 = new Fork();

    new Philosopher(1, F1, F2).start();
    new Philosopher(2, F2, F3).start();
    new Philosopher(3, F3, F4).start();
    new Philosopher(4, F4, F1).start();
  }
}

class Fork { public int num; }
```

```
class Philosopher extends Thread {
  int id;
  Fork F1, F2;
  public Philosopher(int i, Fork f1, Fork f2) {
    this.F1 = f1;
    this.F2 = f2;
    this.id = i;
  }
  public void Dine() {
    System.out.println(id);
  }
  public void run() {
    synchronized (F1) {
      synchronized (F2) {
        Dine();
      }
    }
  }
}
```

Figure 4.8: Dining Philosophers.

threads. Let us consider two elements $x, y \in AH^1_\rho$ such that $T_1 = Tid(ev(x))$ and $T_2 = Tid(ev(y))$ are distinct, the intersection of the lock-sets $ls(x)$ and $ls(y)$ is empty and the acquisition histories in $ah(x)$ and $ah(y)$ are compatible.

Definition 4.3.2 *We say that x and y can be composed when the following conditions hold:*

- $\exists l, l' \subseteq \mathcal{L}$ *s.t.* $l \in ls(x)$ *and* $l' \in ls(y)$
- l' *is in some acquisition history defined in* $ah(x)$ ∎

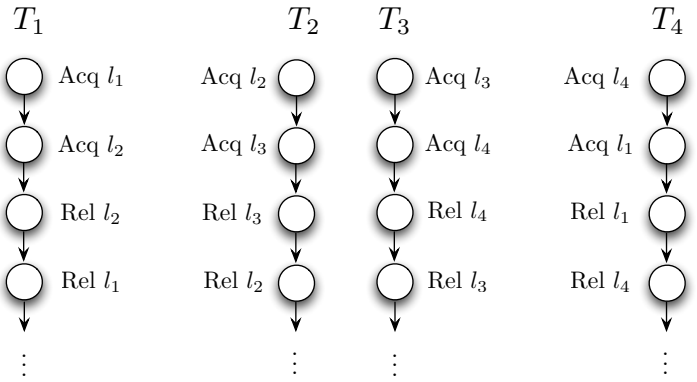


Figure 4.9: Local executions ρ_T for the four threads involved in the Dining Philosophers program of Figure 4.8.

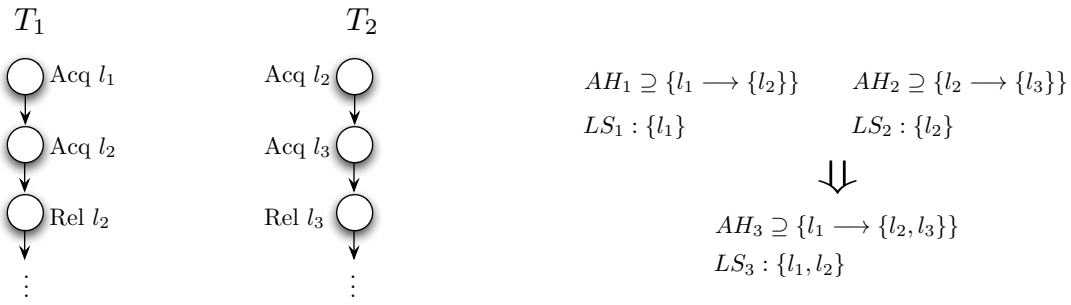


Figure 4.10: Composition of acquisition histories and locks-sets for threads T_1 and T_2.

Before going through the details of the composition, we need to slightly modify the definition of the witnesses set as defined in the previous Section in order to gather composed elements. In particular, the first component of the set AH^1_ρ was the event (T_j, i), where T_j was the thread executing the event. In place of the single thread T_j executing the event now we define a subset of $\mathcal{T}$ containing the thread executing the event. Notice that Theorem 4.3.1 still holds if we adapt the requirement of events executed by different threads to require that $Tid(ev(x)) \cap Tid(ev(y)) = \emptyset$.

When two elements x and $y \in AH^1_\rho$ can be composed the resulting element is $z = ((Tid(ev(x)) \cup Tid(ev(y)), (Occur(ev(x), \rho) + Occur(ev(y), \rho))), ls(x) \cup ls(y), merge(ah(x), ah(y)))$. Figure 4.10 shows how the *merge* procedure, i.e. the composition of acquisition histories, is realized. The intuition behind this is that once we find $x, y \in AH^1_\rho$, with $Tid(ev(x)) = T_1$ and $Tid(ev(y)) = T_2$ respectively, that can be combined; we can assume that all the events executed by T_1 (up to $Occur(ev(x), \rho)$ and T_2 (up to $Occur(ev(y), \rho)$ are executed by a unique thread T'. Then, we collect events, lock-set and acquisition history of such *super thread* T'.

The new set AH^2_ρ is defined as the union of AH^1_ρ and all the elements obtainable by the composition of pairs of elements in AH^1_ρ.

Theorem 4.3.3 *There is a potential deadlocking run $\rho' \in Infer(\rho)$ involving 3 (or 4) threads*

if, and only if,

$\exists x, y \in AH^2_\rho$ such that $Tid(ev(x)) \cap Tid(ev(y)) = \emptyset$, the LockSets $ls(x)$ and $ls(y)$ are disjoint and the acquisition histories in $ah(x)$ and $ah(y)$ are not compatible. ∎

In order to report only one permutation in place of the whole set of permutations representing the same cycle, we add some restrictions to the composition procedure. In particular, $\forall i \in Tid(ev(x))$ and $\forall j \in Tid(ev(y))$ we require $i < j$.

The result can be generalized for potential deadlocking run involving $n > 1$ threads. In particular, in order to detect potential deadlocks involving n threads where $2^{m-1} \leq n \leq 2^m$ for some integer $m > 1$, requires inductively constructing and analyzing the set AH^m_ρ. A deadlock involving n threads is found building $O(log(n))$ sets.

Dining Philosophers In order to explain our algorithm for the prediction of deadlocks involving more than 2 threads we use the dining philosophers problem introduced in the previous Section. At the first round the algorithm generates the set of witnesses AH^1_ρ. No deadlocks are found in this round because $\forall x, y \in AH^1_\rho$ *LockSets* $ls(x)$ and $ls(y)$ are disjoint and the acquisition histories in $ah(x)$ and $ah(y)$ are compatible.

The algorithm then proceeds with the second round, generating AH^2_ρ. AH^2_ρ, contains all the elements of AH^1_ρ and the elements obtained from the composition of T_1 and T_2, T_2 and T_3 and T_3 and T_4. Notice that the elements obtainable from the composition of T_4 and T_1 are discarded by the restriction we introduced. Moreover, no other composition exist.

Among the elements of AH^2_ρ there are $x = ((\{T_1, T_2\}, i), \{l_1, l_2\}, \{l_1 \to \{l_2, l_3\}, l_2 \to \{l_3\}\})$ and $y = ((\{T_3, T_4\}, j)$, $\{l_3, l_4\}, \{l_3 \to \{l_4, l_1\}, l_4 \to \{l_1\}\})$. x and y satisfy the Theorem 4.3.3, so a potential deadlock is found.

At the moment we did not implement the phases II and III for the case when $n > 2$. We have not found practical examples where this was necessary. We did apply prediction algorithms for deadlocks involving $n > 2$ threads (only phase I), but did not detect any potential deadlocks in these benchmarks.

4.4 Schedule Generation Algorithms

Once we generated a cut-point that could expose an atomicity violation or a deadlock, in this Subsection we show how to build a simple execution from scratch from a such cut-point (e, f) that could expose an error, using lock-sets and acquisition histories. We start describing the *theoretically motivated algorithm*. This algorithm assumes that the program is made of n concurrent threads that do not use any mode of communication, but just interact using nested locking; the algorithm is accurate in predicting *under this assumption*.

Despite its promised accuracy, it cannot itself be used in practice because the assumption is false— programs do indeed have causal ordering of events and communication (one thread could wait for another for a signal to proceed), and threads do get created and destroyed, which also gives a causal ordering to events (if T_1 creates T_2, we can't commute events of T_2 before the event in T_1 that created it). Our main idea is to modify the algorithm to make the predicted run *adhere* to the causal ordering of events in the originally observed run as much as possible. The effectiveness of this adherence is not captured in a theoretical assurance, but its proof is in the pudding, as we show that this strategy reduces infeasible executions from being predicted to a large degree in our experiments.

Finally, because of the adherence to the original execution, the predicted executions share the complexity of the original one. In particular, the predicted schedules that we synthesize can have *hundreds of thousands of context-switches*. The number of context-switches affects the time required to schedule the predicted execution, as context-switches take relatively more time than local executions. We provide two heuristics but technically sound transforma-

tions of the schedule that reduce the number of context-switches (the transformations do not break any causal links that may be there in the program).

We now outline the theoretical algorithm [40], the adherence algorithm [40, 91], and the two heuristics to reduce the number of context-switches.

4.4.1 The Theoretical Scheduling Algorithm

Let us, in this Subsection, assume that the program consists of a static set of n threads all already active (view them as n sequential programs interacting with each other), and further assume that the threads do not communicate with each other explicitly using data. Let us assume the threads are $T_1, \ldots, T_n$ and further that e and f occur in threads T_1 and T_2 respectively. Moreover the lock-sets at e and f are disjoint and the acquisition histories at e and f are compatible. Our goal is to find a locking-respecting execution that precisely executes up until e and f in T_1 and T_2.

First, note that, since we assumed there is no communication, executing events from $T_3, \ldots T_n$ can *never help* in the execution of T_1 and T_2 (i.e. never enable events in T_1 and T_2); in fact, they can only hinder finding an execution as they could acquire locks that T_1 and T_2 may require. Consequently, we can completely focus only on scheduling events in threads T_1 and T_2.

For deadlocks we are rescheduling up until a cut-point (e, f) in threads T_1 and T_2, where the event of T_1 that follow e is an acquire of lock l' while holding l. Similarly, the event of T_2 that follow f is an acquire of lock l while holding l'. Similarly, for atomicity violations, the idea is that we are rescheduling up until a cut-point (e, f) in threads T_1 and T_2, where e falls in between e_1 and e_2 in a single thread, and where the sequence e_1 followed by f followed by e_2 realizes the atomicity violation. In doing this alternate schedule, assume that a lock l is held by T_2 at f, and assume that the same lock l is acquired and released by thread T_1, before e. Then we must schedule this block in T_1 *before the last acquire of lock* l *by* T_2 *before* f, as T_1 will have no chance to acquire it once T_2 has made this acquisition.

Computing the above efficiently and in linear time is non-trivial, and this is where the acquisition history helps, as it exactly captures the above information.

Let x_1 and x_2 be the *last* events in T_1 and T_2, respectively, that are before e and f, respectively, with lock-sets empty. Then, we can easily schedule (without violating locking) first T_1 up until x_1 and then T_2 up until x_2. Note that the first events after x_1 (and x_2) either must be e (f) or must be acquisitions of locks that are never released till e (f) is reached. Hence the crux of the scheduling is to schedule from x_1 and x_2 till e and f.

The algorithm works by building a graph of *causal edges* between events. For every lock l in the lock-set of f, if l occurs in the acquisition history of e with respect to some lock l', then we know that after the last acquisition of l' by T_1, there was an acquisition (followed by a release) of the lock l. Hence we know that we must schedule the last release of lock l' in T_1 (say event u) *before* the last acquisition of l in T_2 (say v). We capture this by adding a causal

edge from u to v. Symmetrically, we examine the lock-set of e and the acquisition history of f and throw in causal edges. It turns out that since the acquisition histories are compatible, this graph will by *acyclic*, and hence there is a schedule that respects these orderings. The algorithm simply takes a linearization of partial order to obtain a schedule.

The above argument is adapted from the *proof that the prediction algorithm works correctly*, which appears in Farzan et al. [40], and in turn depends on a proof of a theorem by Kahlon et al. [58]. We have also augmented the above construction by using *vector clocks* to rule out cut-points that are clearly infeasible. We maintain vector-clocks for events that get updated during *thread-creation* and at *barriers* only, and eliminate cut-points (e, f) where e and f are not concurrent with respect to the vector clocks. This greatly reduces the number of infeasible interleavings generated.

4.4.2 Algorithms to Adhere to Original Execution

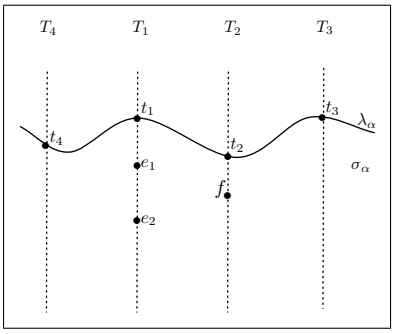
Note that the theoretically synthesized schedule described above blindly executes T_1 till x_1 and T_2 till x_2 (i.e. to the last point where thread have empty lock-sets), and further ignores all other threads. This actually causes the scheduling algorithm to break in practice; when we implemented this, most predicted schedules were not feasible in the program. The reason is that the theoretical assumption that there is a set of n threads from the beginning that do not communicate is not real in practice.

We get around this problem by scheduling a large *prefix* of the run accurately according to the original observed schedule. In this Subsection, we propose a pruning technique for the runs that removes a large prefix of them while maintaining the property that any run predicted from the suffix will still be feasible. The idea behind pruning is to first construct the causal partial order of events of σ, and then remove a set of events from it. The set is a causally prefix-closed set of events (a configuration) that are *causally before* e and f, and where all the locks are free at the end of execution of this configuration. The intuition behind this is that such a configuration can be replayed in the newly predicted execution precisely in the same way as it occurred in the original run, and then stitched to a run predicted from the suffix, since the suffix will start executing in a state where no locks are held.

In order to precisely define this run segment, we define a notion of partial order on the set of events E that captures the causal order. Let D denote the dependency relation between actions defined in 4.2.2.

We define the partial order $\preceq \subseteq E \times E$ on the set of program events as the *least* partial order relation that satisfies the condition that $(e_i, e_j) \in \preceq$ whenever $a_i = \sigma[i]$, $a_j = \sigma[j]$, $i \leq j$, and $(a, a') \in D$ where a_i and a_j are actions performed by events e_i and e_j, respectively.

We identify a causally prefix-closed set of events before e and f to remove. For the cut-point α, define λ_α as the *largest* subset of events of σ that has the following properties: (1) it does not contain e or f, (2) for any event e' in λ_α, all events $e'' \preceq e'$ are in λ_α, and (3) for any event e' in T_i such that e' is the last event of T_i in λ_α (with respect to $\sqsubseteq_i$), the lock-set associated to e' in T_i is empty. In the above figure, the curve labeled λ_α marks the boundary of λ_α, and events $T_1, \ldots, T_4$ have empty lock-sets.



The run segment relevant to a cut-point α is then defined as the set of events in $\sigma_\alpha = \sigma \setminus \lambda_\alpha$ scheduled according to the total order in σ ($\leq$).

Similarly to what described in the previous Subsection, the algorithm works by building a graph of *causal edges* between events of σ_α. The algorithm simply takes a linearization of partial order to obtain a schedule.

4.4.3 Heuristic for Reducing Context-switches

While the above described algorithms focus on generating executions that are likely to be *feasible*, they ignore the overhead complexity of actually scheduling the runs. The executions we observe have a large number of context-switches (hundreds of thousands in some of our examples), and since our predicted executions try to adhere to the observed runs for feasibility, they also have a large number of context switches. We hence employ two heuristics to bring down the number of context-switches, while at the same time preserving feasibility. Both heuristics rely on the idea that if a sequence of actions is provably non-interfering among the threads, then we can execute the sequence in any order, and in particular, execute them with the least number of context-switches.

The two methods below essentially reshuffle the predicted execution (very conservatively) preserving feasibility but reducing the number of context-switches (usually by about 10% to 20% in the experiments). The algorithms to achieve this are simple 1-pass processing of the schedules, and helps reduce the time taken to reschedule executions.

4.4.3.1 Escape Analysis to Reduce Context-switches

The first idea is to observe which of the (non-local) accesses to memory locations are actually shared in the run; in actual runs, several shared variables may be touched by only one thread. Let us call these memory locations *unshared* in the observed run. Unshared variables are not known in advance, and we do observe them, and they do play a role when we try to reschedule executions. However, accesses to unshared variables by a thread cannot affect another thread, and hence we can shuffle these to obtain fewer context-switches, without affecting feasibility.

4.4.3.2 Identifying Read-blocks to Reduce Context-switches

The second idea is that even if there are a large number of contiguous *reads* to shared variables, these accesses do not cause real interference between the threads. Hence the reads in this contiguous block can be shuffled in any way without affecting feasibility of the schedule. Again, we schedule these reads in a way that minimizes the number of context-switches, and reduce it to at most the number of active threads in the block.

Chapter 5

Precise and Relaxed Prediction: SMT Solver Based Approach

The crux of the previous Chapter is the prediction algorithm, which reasons at an abstract level in order to efficiently but accurately predict atomicity-violating and deadlocking schedules.

The prediction algorithm was based on *lock-sets* and *acquisition histories*, which only ensure that the predicted run respects the *lock* acquisitions and releases in the run. In other words, the predicted runs are certainly not guaranteed to be feasible in the original program— if the original program had threads that communicated through shared variables in a way that orchestrated the control flow, the predicted runs may just not be feasible.

In this Chapter we present a fundamentally new algorithm that uses logical reasoning and the fast SMT solvers available today to predict runs accurately. In order to present such algorithm we explore a new target for predictive testing of concurrent programs that is fundamentally very different from deadlocks, data-races or atomicity errors: we propose to target executions that lead to *null-pointer dereferences*. Given an arbitrary execution of a concurrent program under test, we investigate fundamental techniques to accurately and scalably predict executions that are likely to lead to null-pointer dereferences.

Null-pointer dereferences can occur in a thread when dereferencing local variables. Consequently, an *accurate* prediction of null-pointer dereferences requires, by definition, handling local variables and the computation of threads. This is in sharp contrast to errors like data-races and atomicity violations, which depend only on *accesses to shared variables*.

The prediction algorithm aims to find some interleaving of the events in the observed run that will result in a null-pointer dereference. The naive approach to this problem is to reduce it to a constraint satisfaction problem; the set of constraints capture the semantics of local computations as well as the interaction of threads using reads and writes, concurrency control like locks, etc. A constraint solver could then solve these constraints and hence synthesize an interleaving that causes a null-pointer dereference to occur (this is similar to logic-based bounded model-checking for concurrent programs [87, 88]). There are however several challenges in such a naive formulation. First, decidable logical constraint solvers can only handle very restricted logical fragments like linear arithmetic, uninterpreted function theories, and basic theories of arrays [32]; if the constraints generated do not fit within these theories, then using these solvers is impossible. For example, a simple statement in the program $x := y * z$ will introduce non-linear arithmetical

constraints that do not fit into any decidable logical theory. Getting around this can be done by cutting corners (e.g. concretizing values or modeling multiplication as an uninterpreted function), but this will immediately make runs infeasible. Secondly, the runs over which we work with involve tens of thousands to millions of instructions. Even the fastest SMT solvers available today have no hope of handling them. Consequently, accurate null-pointer dereference prediction seems intractable.

We propose a new technique for prediction that we call *computation agnostic logical prediction*. The key idea of this approach is to *completely* avoid the modeling of individual instructions of the concurrent program under test, but model accurately and completely the communication across shared variables as well as concurrency synchronization. Given an execution of a program under test, we look upon the local instructions of the threads, that occur between shared accesses, as *black-boxes* that somehow captures the effect of the code on the the next write to a shared variable. We then ask, using logic, whether there is a way to *stitch the black-boxes* in such a way as to expose bugs (e.g. null-pointer dereferences, violate atomicity, etc.), while maintaining the expected communication on shared variables that these black-boxes expect. More precisely, we monitor the values of shared variables and ensure that each black-box sees precisely the same value of shared variables as it did in the original run, and hence will compute the same values on the shared variables as it did in the original run. The logical constraints also ensure that the predicted run satisfies the semantics of concurrency synchronization primitives such as locks, barriers, thread-creation, etc., that obviously need to be respected.

In Chapter 7 we show the remarkable result that despite this coarse modeling, we are always *guaranteed* that the predicted runs will always be feasible in the program. Furthermore, stitching black-boxes and ignoring the computation inside the black-box gives far simpler logical constraints. Since the constraints do not capture computation, complex instructions in the program pose no problems. In fact our logical constraints fall into a decidable logic fragment, the quantifier-free fragment of *difference logic*. Difference logic constraints express only ordering constraints between integers, and are Boolean combinations of constraints of the form $x - y \leq c$. This logic is very efficiently decidable, and the current SMT solvers (like Z3 [32]) scale extremely well to handle the constraints we construct from testing. This results in a fast accurate prediction algorithm.

Accurate prediction, though good in theory and fast, does not work well in practice because the constraint solver many a time finds *no solution* to the constraints. Demanding that the predicted schedule be absolutely guaranteed to be feasible seems too stringent a requirement, especially given that we treat code as black-boxes. At the end of the Chapter we propose a relaxed prediction algorithm, where we explore, for an increasing threshold k, whether there is a predictable schedule that violates at most k constraints in the accurate prediction formulation. As we show in experiments, even k being 1 or 2 is sufficient to pass the prediction phase for most runs, yielding a predicted run that is not absolutely guaranteed to be feasible but perhaps very close to being feasible as only k clauses have been violated.

We show empirically in Chapter 7 that most of these predicted interleavings are actually entirely feasible, and are far better than the feasibility of arbitrary runs that we synthesized using the algorithms presented in Chapter 4.

The Chapter is organized as follow. We introduce the prediction model in Section 5.1, then we introduce the problem of predicting null-pointer dereferences that will be used as driving example in this Chapter. In Sections 5.3 and 5.4 we present the algorithms for the precise prediction by logical constraint solving and to how to increase the scalability of the proposed approach respectively. In Section 5.5 we introduce the relaxed prediction algorithm and finally in Section 5.6 we describe how to extend the precise prediction algorithm to other concurrency errors, i.e. atomicity-violating schedules and data-races.

5.1 Prediction Model

We now set up the formal notation to describe the run prediction phase. We model the runs of a concurrent program as a *word* where each letter describes the action done by a thread in the system (as described earlier in Chapter 2). The word will capture the essentials of the run— shared variable accesses, synchronizations, thread-creating events, etc. However, we will *suppress* the local computation of each thread, i.e. actions a thread does by manipulating local variables, etc. that are not (yet) visible to other threads, and model the local computation as a *single* event *lc*.

We fix an infinite countable set of thread identifiers $\mathcal{T}$= $\{T_1, T_2, ..\}$ and define an infinite countable set of shared variable names SV that the threads manipulate. We also fix a countable infinite set of locks $\mathcal{L}$.

The actions that a thread T_i can perform on a set of shared variables SV and global locks $\mathcal{L}$ is defined as:

$$\Sigma_{T_i} = \{T_i{:}\,\mathit{read}_{x,val},\ T_i{:}\,\mathit{write}_{x,val} |\ x \in SV, val \in \mathit{Val}(x)\} \cup \{\ T_i{:}\,\mathit{lc}\} \cup \{T_i{:}\,\mathit{acquire}(l),\ T_i{:}\,\mathit{release}(l) |\ l \in \mathcal{L}\}$$
$$\cup \{T_i{:}\ tc\ T_j \mid T_j \in \mathcal{T}\}$$

We define $\Sigma = \bigcup_{T_i \in T} \Sigma_{T_i}$ as the set of actions of all threads. A word w in Σ^*, in order to represent a run, must satisfy several obvious syntactic restrictions as Lock-validity, Data-validity, and Creation-validity. Notice that respect to the Chapter 4 we now require that w has to satisfy Data-validity. That is, a run should respect not only the semantics of locks and thread creation but also the semantic of reads i.e. whenever a read of a value from a variable occurs, the last write to the same variable must have written the same value.

5.1.1 The Maximal Causal Model for Prediction

Given a run σ corresponding to an actual execution of a program P, we would like our prediction algorithms to synthesize new runs that interleave the events of σ to cause an error. However, we want to predict *accurately*; in other

words we want the predicted runs to be feasible in the actual program.

We now give a sufficient condition for a partial run predicted from an observed run to be *always* feasible. This model of prediction was defined by Şerbănuţă et al., and is called the *maximal causal model* [85]; it is in fact the most liberal prediction model that ensures that the predicted runs are always feasible in the program that work purely dynamically (i.e. no other information about the program is known other than the fact that it executed this set of observable events, which in turn do not observe computation). We generalize the model slightly by taking into account thread creation.

Definition 5.1.1 (Maximal causal model of prediction [85]) *Let σ be a run over a set of threads $\mathcal{T}$, shared variables SV, and locks $\mathcal{L}$. A run σ' is* precisely predictable *from σ if (i) for each $T_i \in T$, $\sigma'|_{T_i}$ is a prefix of $\sigma|_{T_i}$, (ii) σ' is lock-valid, (iii) data-valid, and (iv) creation-valid. Let $PrPred(\sigma)$ denote the set of all runs that are precisely predictable from the run σ.* ∎

The first condition above ensures that the events of T_i executed in σ' is a prefix of the events of T_i executed in σ. This property is crucial as it ensures that the local state of T_i can evolve correctly. Note that we are forcing the thread T_i to read the same values of global variables as it did in the original run. Along with data-validity, this ensures that the thread T_i reads precisely the same global variable values and updates the local state in the same way as in the original run. Lock-validity and creation-validity are, of course, required for feasibility. We will refer to runs predicted according to the maximal causal model (i.e. runs in $PrPred(\sigma)$) as the precisely predicted runs from σ.

The following soundness of the prediction that assures all predicted runs are feasible, follows:

Theorem 5.1.2 ([85]) *Let P be a program and σ be a run corresponding to an execution of P. Then every precisely predictable run $\sigma' \in PrPred(\sigma)$ is feasible in P.* ∎

Proof: If $\rho'|_{T_i}$ is a prefix of $\rho|_{T_i}$ for each $T_i \in T$ implies that the set of events in ρ' is a subset of events in ρ. Furthermore, the program order is respected by the threads in ρ'. Suppose that there is a global computation gc followed by a local computation lc in $\rho'|_{T_i}$. It is implied that gc is followed by lc in $\rho|_{T_i}$ as well. Now, there might be several cases according to what gc is and we argue that in all cases the local computation lc is unaffected and is the same as in run ρ. If gc is a write to variable x, i.e. $gc \in \{T_i : write_{x,val} \mid x \in SV \text{ and } val \in Val(x)\}$ then obviously it does not affect the local computation lc. Same thing is true when gc correspond to a lock event, i.e. $gc \in \{T_i : write_{x,val} \mid x \in SV \text{ and } val \in Val(x)\} \cup \{T_i\text{:}\,\textit{acquire}(l),\ T_i\text{:}\,\textit{release}(l) \mid l \in L\}$. The only case when gc can affect local computation lc is when gc is a read event, say reading value val from variable x, and lc is using the value val in its computation. Note that by the first condition gc is forced to read value val from variable x and by data-validity we know that the most recent write to x in ρ' before gl would provide value val. Therefore, gc cannot

read any value other than val in ρ' and hence local computation lc in ρ' would remain the same as in ρ. From the feasibility of the original run ρ along with the lock-validity of ρ' we can conclude that ρ' is feasible. ∎

The above theorem is independent from the class of programs. We will assume however that the program is locally deterministic (non-determinism caused by threads interleaving is, of course, allowed). The above theorem, in fact, even holds when local computations of P are *non-deterministic*; i.e. the predicted runs will still be feasible in the program P. However, in order to be able to execute the predicted runs, we need to assume determinism of local actions. In this case, we can schedule the run σ' precisely and examine the outcomes of the tests on these runs.

5.2 Prediction Problem for Null-reads

We are now ready to formally define the precise prediction problem for forcing null-reads.

Definition 5.2.1 (Precisely predictable null-reads) *Let σ be a run of a program P. We say that σ' is a* precisely predictable run that forces null-reads *if there is a thread T_i and a variable x such that the following are satisfied: (i) $\sigma' = \sigma''.f$ where f is of the form T_i:read$_{x,null}$, (ii) σ'' is a precisely predictable run from σ using the maximal causal model, and (iii) there is some val $\neq$ null such that $(\sigma''|_{\Sigma_i})$. T_i:read$_{x,val}$ is a prefix of $\sigma|_{\Sigma_i}$.* ∎

Intuitively, the above says that the run σ' must be a precisely predictable run from σ followed by a read of null by a thread T_i on variable x, and further, in the observed run σ, thread T_i must be executing a non-null read of variable x after executing its events in σ''. The above captures the fact that we want a precisely predictable run followed by a single null-read that corresponded to a non-null read in the original observed run. Note that σ' itself is not in $PrPred(\sigma)$, but is always feasible in the program P, and results in a null-read by thread T_i on variable x that had not happened in the original run.

The precisely predictable runs that force null-reads are hence excellent candidates to re-execute and test; if the local computation after the read does not check the null-ness of x before dereferencing a field of x, then this will result in an exception or error.

5.2.1 Motivating Example

We motivate the appeal of targeting null-reads as a way of finding null-pointer dereferencing executions, and the concept of predictions based on *global stitching* using an example.

Consider a code extract from the `Pool 1.2` library [8] in the `Apache Commons` collection, presented in Figure 5.1. The object `pool`'s state, *open* or *closed*, is tested outside the synchronized block in method `returnObject`, by checking whether the flag variable `isClosed` is true. If so, then some local computation occurs, followed by a

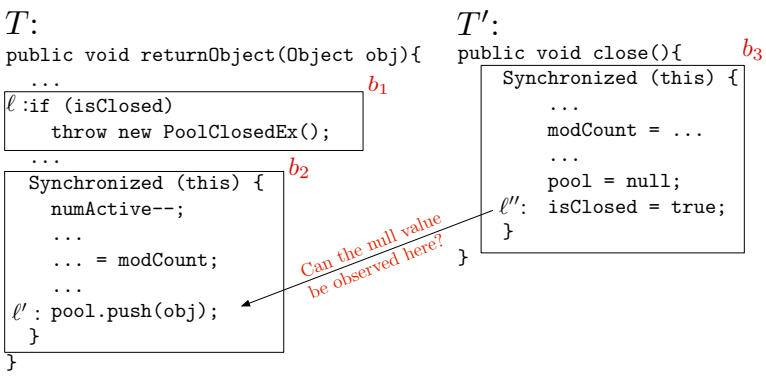


Figure 5.1: Code snippet of the buggy implementation of Pool.

synchronized block that dereferences the shared object `pool`. A second method `close` closes the pool and sets `isClosed` to true to signal that the pool has been closed.

An error in this code (and such errors are very typical) stems from the fact that the check of `isClosed` in the method `returnObject` is not within the synchronized block; hence, if a thread executes the check at line ℓ, and then a concurrent thread executes the method `close()` before the synchronized block begins, then the access to the pool object at line ℓ' will raise an uncaught *null-pointer dereference exception*.

In a dynamic testing setting, consider the scenario where we observe an execution σ with two threads, where T executes the method `returnObject` first, and then, T' executes the method `close` after T finishes executing `returnObject`. There is no null-pointer dereference in σ. Our goal is to predict an alternate scheduling of events of σ that causes a null-pointer dereference.

Our prediction for null-pointer dereferences works as follows. In the run σ, a read of the shared variable `pool` at ℓ' occurs in T and the read value is not null. Also, a write to `pool` occurs in T' at ℓ'' which writes the value `null`. We ask whether there exists an alternative run σ' in which, the read at ℓ' (in T) can read the value `null` written by the write at location ℓ'' (in T') (as illustrated by the arrow in Figure 5.1).

Our prediction algorithm observes the shared events (such as shared reads/writes) but suppresses the semantics of local computations entirely and does not even observe them; they have been replaced by "..." in the figure as they play no role in our analysis.

Prediction of runs that force the read at ℓ' to read the null value written at ℓ'' must meet several requirements. Even if the predicted run respects the synchronization semantics for locks, thread creation, etc., the run may diverge from the observed run due to reading a different set of values for shared variables which will result in a different local computation path (e.g. the condition check at ℓ will stop the computation of the function right away if the value of `isClosed` is true). Therefore, we also demand that all other shared variable reads read the same value as they did

in the original observed run, in order to guarantee that unobserved local computations will unfold in the same way as they did in the original run. This ensures the feasibility of the predicted runs.

5.2.2 Identifying *null-WR* pairs Using Lock-sets

The first phase of our prediction is to identify *null-WR* pairs $\alpha = (e, f)$ where e is a write of null to a variable and f is a read of the same variable, but where the read in the original run reads a non-null value. Moreover, we would like to identify pairs that are feasible at least according to the hard constraints of thread-creation and locking in the program. For instance, if a thread writes to a shared variable x and reads from it in the same lock-protected region of code, then clearly the read cannot match a write protected by the same lock in another thread. Similarly, if a thread initializes a variable x to a non-null and then creates another thread that reads x, clearly the read cannot read an uninitialized x. We use a lock-set based static analysis (introduced in Chapter 4) of the run (without using a constraint solver) to filter out such impossible read-write pairs. The ones that remain are then subject to a more intensive analysis using a constraint solver.

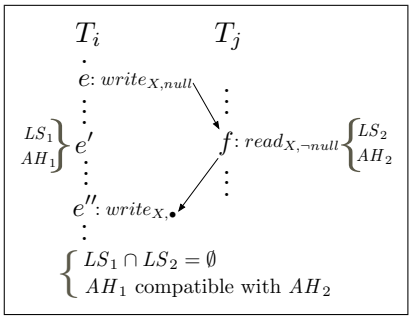
Using a static analysis on the observed run σ, we first collect all *null-WR* pairs $\alpha = (e, f)$. Then, we prune away *null-WR* pairs for which there is no lock-valid run in which f is reading the null value written by e. Then, for each *null-WR* pair $\alpha = (e, f)$ left, we use our precise logical prediction algorithm to obtain a lock-valid, data-valid and creation-valid run in which f is reading the null value written by e. However, instead of using run σ for the purposes of the prediction, we slice a *relevant* segment of it, and use the segment instead. The slicing algorithm is build on top of that one introduced in Section 4.4.2. The reason for this is twofold: (1) these run segments are often orders of magnitude smaller than the complete run, and this increases the scalability of our technique and (2) when a precisely predictable run does not exist, we use a more relaxed version of the constraints to generate a new run, limiting the improvisation to a smaller part of run increases our chances of obtaining a feasible execution. We formally define this *relevant run segment* and how it is computed in Section 5.4.

In this static analysis, the idea is to check if the *null-WR* pair $\alpha = (e, f)$ can be realized in a run that respects lock-validity and creation-validity constraints only (and not data-validity). Creation validity is captured by computing a vector clock associated with each event, where the vector clock captures only the hard causality constraints of thread creation. If f occurs before e according to this relation, then clearly it cannot occur after e and the pair is infeasible. Lock-validity is captured by reducing the problem of realizing the pair (e, f) to *pairwise reachability* under nested locking [58], which is then solved by computing lock-sets and acquisition histories for each event. Similar techniques have been exploited for finding *atomicity* violations in the Section 4.2.

5.2.3 Checking Lock-valid Reachability

Consider a *null-WR* pair $\alpha = (e, f)$ and a run σ in which f (a read in thread T_j) occurs first, and later the write event e is performed by T_i.

Let us assume that e'' is the next write event (to the same variable accessed in e and f) in T_i after e. If there exists a lock-valid run σ' (obtained by permuting the events in σ) in which f reads the null value provided by e, then in σ', f should be scheduled after e, but before e''; if f is scheduled also after e'', then the write in e'' overwrites the null value written by e before it reaches f. This means that there should exist an event e' of thread T_i, occurring between events e and e'', that is executed right before (or after f) in σ'; in other words, e' and f are co-reachable.



As we have done in Section 4.2, we check if there is an event e' in T_i between e and e'' such that e' and f are co-reachable in a lock-valid run. The co-reachability check is done by examining the lock-sets and acquisition histories at e' and f: the lock-sets at e' and f must be disjoint and the acquisition histories at e' and f must be compatible.

Note that the above condition is *necessary* for the existence of the lock-valid run σ', but not *sufficient*; hence filtering out pairs that do not meet this condition is sound.

5.3 Precise Prediction by Logical Constraint Solving

We now describe how we solve the problem of precisely predicting a run that realizes a *null-WR* pair $\alpha = (e, f)$. This problem is to predict whether there is an alternate schedule in the maximal causal model that forces the read at f to read the null value written by e. We solve this using a logic constraint solver (an SMT solver); the logic of the constraints is in a fragment that is efficiently decidable.

Prediction according to the maximal causal model is basically an encoding of the creation-validity, data-validity, and lock-validity constraints using logic, where quantification is removed by expanding over the finite set of events under consideration. Modeling this using constraint solvers has been done before ([80]) in the context of finding data-races. We reformulate this encoding briefly here for several reasons. First, this makes the exposition self-contained, and there are a few adaptations to the null-read problem that need explanation. Second, we perform a wide set of carefully chosen optimizations on this encoding, whose description needs this exposition. And finally, the relaxation technique, is best explained by referring directly to the constraints.

$$\psi \equiv PO \wedge CV \wedge DV \wedge LV$$

$$PO = (\textstyle\bigwedge_{i=1}^{n} PO_i) \wedge C_{init}$$

$$C_{init} = \textstyle\bigwedge_{i=1}^{n} (ts_{e_{init}} < ts_{e_{i,1}})$$

$$PO_i = \textstyle\bigwedge_{j=1}^{m_i - 1} (ts_{e_{i,j}} < ts_{e_{i,(j+1)}})$$

$$DV = \textstyle\bigwedge_{x} \quad \bigwedge_{val \in Val(x)} \quad \bigwedge_{r \in R_{x,val}} \left(\bigvee_{w' \in W_{x,val}} Coupled_{r,w'} \right)$$

$$Coupled_{r,w} = (ts_w < ts_r) \wedge \bigwedge_{e'' \in W_x - \{w\}} ((ts_{e''} < ts_w) \vee (ts_r < ts_{e''}))$$

$$LV = LV_1 \wedge LV_2$$

$$LV_1 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{[e_{ac}, e_{rel}] \in L_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e_{rel}} < ts_{e'_{ac}} \vee\ ts_{e'_{rel}} < ts_{e_{ac}} \right)$$

$$LV_2 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{e_{ac} \in NoRel_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e'_{rel}} < ts_{e_{ac}} \right)$$

Figure 5.2: Constraints capturing the maximal causal model.

5.3.1 Capturing the Maximal Causal Model Using Logic

Given a run σ, we first encode the constraints on all runs predicted from it using the maximal causal model, independent of the specification that we want runs that match a given *null-WR* pair. A predicted run can be seen as a total ordering of the set of events E of the run σ. We use an integer variable ts_e to encode the *timestamp* of event $e \in E$ when e occurs in the predicted run. Using these timestamps, we logically model the constraints required for precisely predictable runs (see Definition 5.1.1), namely that the run respect the program order of σ, that it be lock-valid, data-valid, and creation-valid.

Figure 5.2 illustrates the various constraints. The constraints are a conjunction of program order constraints (PO), creation-validity constraints (CV), data-validity constraints (DV), and lock-validity constraints (LV) (please refer to Section 2.2.2).

The program order constraint (PO) captures the condition that the predicted run respect the program order of the original observed run. Suppose that the given run σ_α consists of n threads, and let $\sigma_\alpha|_{T_i}$ = $e_{i,1}$, $e_{i,2}$, ..., e_{i,m_i} be the sequence of events in σ_α that relates to thread T_i. Then the constraint PO_i demands that the time-stamps of the predicted run obey the order of events in thread T_i, and PO demands that all threads meet their program order. We also consider an initial event e_{init} which corresponds to the initialization of variables. This event should happen before any thread starts the execution in any feasible permutation, and is encoded as the constraint C_{init}.

Turning to creation-validity, suppose that $e_{tc(i)}$ is the event that creates thread T_i. Then the constraint CV demands

that the first event of T_i can only happen after $e_{tc(i)}$. Combined with program order constraint, this means that all events before the creation of T_i in the thread that created T_i must also occur before the first event of T_i.

The data-validity constraints DV capture the fact that reads must be coupled with appropriate writes; more precisely, that every read of a value from a variable must have a write before it writing that value to that variable, and moreover, there is no other intermediate write to that variable. Let $R_{x,val}$ represent the set of all read events that read value val from variable x in σ_α, W_x represent the set of all write events to variable x, and $W_{x,val}$ represent the set of all write events that specifically write value val to variable x. For each read event $r = read_{x,val}$ and write event $w \in W_{x,val}$, the formula $Coupled_{r,w}$ represents the requirement that w is the most recent write to variable x before r and hence r is coupled with w. The constraint DV demands that all reads be coupled with writes that write the same value as the read reads.

Lock-validity is captured by the formula LV. We assume that each lock acquire event ac of lock l in the run is matched by precisely one lock release event rel of lock l in the same thread, unless the lock is not released by the thread in the run. We call the set of events in thread T_i between ac and rel a lock block corresponding to lock l represented by $[ac, rel]$. Let $L_{i,l}$ be the set of lock blocks in thread T_i regarding lock l. Then LV_1 asserts that no two threads can be simultaneously inside a pair of lock blocks $[e_{ac}, e_{rel}]$ and $[e'_{ac}, e'_{rel}]$ corresponding to the same lock l. Turning to locks that never get released, the constraint LV_2 handles asserts that the acquire of lock l by a thread that never releases it must always occur after the releases of lock l in every other thread. In this formula, $\mathit{NoRel}_{i,l}$ stands for lock acquire events in T_i with no corresponding later lock release event.

5.3.2 Optimizations

The constraints, when written out as above, can be large. We do several optimization to control the formula bloat (while preserving the same logical constraint).

The data-validity constraint above is expensive to express, as it is, in the worst case, cubic in the maximum number of accesses to any variable. There are several optimizations that reduce the number of constraints in the encoding. Suppose that $r = read_{x,val}$ is performed by thread T_i.

- Each write event w' to x that occurs after r in T_i, i.e. $r \sqsubseteq_i w'$, can be excluded in the constraints related to coupling r with a write in constraint DV above.
- Suppose that w is the most recent write to x before r in T_i. Then, each write event w' before w in T_i, (i.e. $w' \sqsubseteq_i w$), can be excluded in the constraints related to coupling r with a write in constraint DV above.
- When r is being coupled with $w \in W_{x,val}$ in thread T_j, each write event w' before w in T_j, i.e. $w' \sqsubseteq_j w$, can be excluded as candidates for e'' in the formula $\mathit{Coupled}_{r,w}$.

- Suppose that r is being coupled with $w \in W_{x,val}$ in thread T_j and w' is the next write event to x after w in thread T_j. Then each write event w'' after w' in T_j, i.e. $w' \sqsubseteq_j w''$, can be excluded as candidates for e'' in the formula $\mathit{Coupled}_{r,w}$.
- Event r can be coupled with e_{init} only when there is no other write event to x before r in T_i, i.e. $\nexists w.\ (w \sqsubseteq_i r \wedge w \in W_x)$. Furthermore, it is enough to check that the first write event to x in each thread (if it exists) is performed after r.

The lock-validity formula above, which is quadratic in the number of lock blocks, is quite expensive in practice. We can optimize the constraints. If a read event r in thread T_i can be coupled with only *one* write event w which is in thread T_j then in all precisely predictable runs, w should happen before r. Therefore, the lock blocks according to lock l that are in T_j before w and the lock blocks according to lock l that are in T_i after r are already ordered. Hence, there is no need to consider constraints preventing T_i and T_j to be simultaneously in such lock blocks. In practice, this greatly reduces the number of constraints. Furthermore, when considering lock acquire events with no corresponding release events in LV_2 above, it is sufficient to only consider the *last* corresponding lock blocks in each thread and exclude the earlier ones from the constraint.

5.3.3 Predicting Runs for a *null-WR* pair

We adapt the above constraints for predicting in the maximal causal model to predict whether a *null-WR* pair $\alpha = (e, f)$ is realizable. Suppose that σ and $\alpha = (e, f)$ are a run and a *null-WR* pair passed to the prediction phase, respectively. Notice that in the original run f reads a non-null value while we will force it to read null in the predicted run by coupling it with write event e. Indeed, this is the whole point of predicting runs— we would like to diverge from the original run at f by forcing f to read a null value. Note that once f reads a different value, we no longer have any predictive power on what the program will do (as we do not examine the code of the program but only its runs). Consequently, we cannot predict any events causally later than f.

The prediction problem is hence formulated as follows:

Given a run σ, and a null-WR pair $\alpha = (e, f)$ in σ, algorithmically find a precisely predictable run from σ that forces null-reads according to α; i.e. f is the last event and reads the null value written by e.

The prediction problem is to find precisely predicted runs that execute e followed by f, while avoiding any other write to the corresponding variable between e and f. The constraints that force the read f be coupled with the write e is NC = $Coupled_{f,e}$.

Furthermore, recall that the feasibility of the run that we are predicting needs to be ensured only *up to* the read f. Consequently, we drop from the data-validity formula that the value read at f (in the original run) match the last write (it should instead match e as above).

A further complication is scheduling events that happen after e in the same thread. Note that some of these events may need to occur in order to satisfy the requirements of events before f (for instance a read before f may require a write after e to occur). However, we may not want to predict some events after e, as we are really only concerned with f occurring after e. Our strategy here is to let the solver figure out the precise set of events to schedule after e (and before the next write to the same variable as e is writing to) in the same thread.

For events after e in T_i, we enforce lock-validity and data-validity constraints only if they are scheduled *before* f. More precisely, we replace $\vee_{w'} Coupled_{r_i,w'}$ in the formula DV to $(ts_r < ts_f \Rightarrow \vee_{w'} Coupled_{r_i,w'})$. Similarly, we drop the lock constraints on events occurring after f (this relaxation is more involved but straightforward).

In summary, we have reduced the problem of predicting a run according to the maximal causal model that causes the null write-read pair to be realizable to a satisfiability of a formula ψ in logic. The constraints generated fall within the class of quantifier-free difference logic constraints which SMT solvers efficiently solve in practice.

5.4 Pruning Executions for Scalability

Identifying *null-WR* pairs using the lock-set based analysis and then subjecting them to constraint checking is a precise method to force null reads. However, in our experiments, we discovered that the constraint solving approach does not scale well when runs get larger. In this Section, we propose a pruning technique for the runs that removes a large prefix of them while maintaining the property that any run predicted from the suffix will still be feasible. While this limits the number of predictable runs in theory, we show that in practice, it does not prevent us from finding errors (in particular, *no error* was missed due to pruning in our experiments). Furthermore, in Chapter 7 we show that in practice pruning improves the scalability of our technique, in some cases by an order of magnitude.

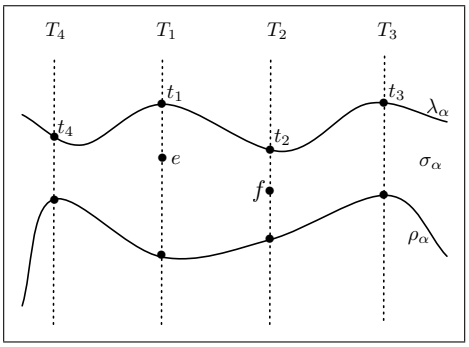
Consider an execution σ and a *null-WR* pair $\alpha = (e, f)$. The idea behind pruning is to first construct the causal partial order of events of σ, and then remove two sets of events from it. The first set consists of events that are *causally after* e and f (except for some events, as described in detail below). The second set is a causally prefix-closed set of events (a configuration) that are *causally before* e and f, and where all the locks are free at the end of execution of this configuration. The intuition behind this is that such a configuration can be replayed in the newly predicted execution precisely in the same way as it occurred in the original run, and then stitched to a run predicted from the suffix, since the suffix will start executing in a state where no locks are held.

In order to precisely define this run segment, we define a notion of partial order on the set of events E that captures the causal order. Let D denote the dependency relation between actions that relates two actions of the same thread, reads and writes on the same variable by different threads, and lock acquisition and release actions of the same lock in different threads. We define the partial order $\preceq \subseteq E \times E$ on the set of program events as the *least* partial order relation

that satisfies the condition that $(e_i, e_j) \in \preceq$ whenever $a_i = \sigma[i]$, $a_j = \sigma[j]$, $i \leq j$, and $(a, a') \in D$ where a_i and a_j are actions performed by events e_i and e_j, respectively.

Let us define ρ_α as the *smallest* subset of events of σ that satisfies the following properties: (1) ρ_α contains events e and f, (2) for any event e' in ρ_α, all events $e'' \preceq e'$ are in ρ_α, and (3) for every event corresponding to a lock $acquire$ in ρ_α, its corresponding $release$ event is also in ρ_α.

The intuition is that events that are not in ρ_α are not relevant for the scheduling of the *null-WR* pair ; they are either far enough in the future, or are not dependent on any of the events in ρ_α. The figure below presents a run of a program with 4 threads that is projected into individual threads. Here, e belongs to thread T_1 and f belongs to thread T_2. The cut labeled ρ_α marks the boundary after which all events are not *causally before* e and f, and hence, need not be considered for the generation of the new run.



Next, we identify a causally prefix-closed set of events before e and f to remove. For the *null-WR* pair α, define λ_α as the *largest* subset of events of ρ_α that has the following properties: (1) it does not contain e or f, (2) for any event e' in λ_α, all events $e'' \preceq e'$ are in λ_α, and (3) for any event e' in T_i such that e' is the last event of T_i in λ_α (with respect to $\sqsubseteq_i$), the lock-set associated to e' in T_i is empty. In the above figure, the curve labeled λ_α marks the boundary of λ_α, and events $T_1, \ldots, T_4$ have empty lock-sets.

The run segment relevant to a *null-WR* pair α is then defined as the set of events in $\sigma_\alpha = \rho_\alpha \setminus \lambda_\alpha$ scheduled according to the total order in σ ($\leq$). One can use a simple worklist algorithm to compute both ρ_α and λ_α, and consequently σ_α. This run segment is passed to the run prediction phase, in the place of the whole run σ.

5.5 Relaxed Prediction

The encoding proposed in the previous Section is sound, in the sense that it guarantees feasibility of the predicted runs. However, sound prediction under the maximal causal model can be too restrictive and result in predicting no runs. Slightly diverging from the original can sometimes lead to prediction of runs that are feasible in the original program.

For instance, in the example 5.1, the variable `modCount` is a global counter keeping track of the number of modifications made to the `pool` data structure, and does not play any role in the local control flow reaching the point of null-pointer dereference at ℓ''. In the real execution leading to this null-pointer dereference, which is the one where block b_1 (from T) is executed first, followed by b_3 (from T') and then b_2 (from T), the read of `modCount` will read

a different value than the corresponding value read in σ. However, this does not affect the feasibility of the run (in contrast to the value read for `isClosed`, which plays an important role in reaching the null-pointer dereference).

The ideal algorithm will find an execution that violates the read condition on `modCount`, but yet finds a feasible run that causes the null-pointer dereference in this example.

We hence have a tension between two choices— we would like to maintain the same values read for as many shared variable reads as possible to increase the probability of getting a feasible run, but at the same time allow a few reads to read different values to make it possible to predict some runs. Our proposal, which is one of the main contributions of this paper, is an iterative algorithm for finding the *minimum* number of reads that can be exempt from data-validity constraints that will allow the prediction algorithm to find at least one run. We define a suitable *relaxed* logical constraint system to predict such a run. Our experiments show that exempting a few reads from data-validity constraints greatly improves the flexibility of the constraints and increases the possibility of predicting a run, and at the same time, the predicted runs are often feasible.

The iterative algorithm works as follows. Let's assume there are n shared variable reads that are required to be coupled with specific write by the full set of data-validity constraints. The data-validity constraints are expressed so that we specifically ask for n shared reads to be coupled correctly. If we fail to find a solution satisfying constraints for all n reads, then we repeatedly decrement n, and attempt to find a solution that couples $n-1$ reads in the next round, and so on. The procedure stops whenever a run (solution) is found. The change required in the encoding to make this possible is described below.

For every read event $r_i \in R$, we introduce a new Boolean variable, b_i, that is true if the data-validity constraint for r_i is satisfied, and false otherwise. In addition, we consider an integer variable $bInt_i$ which is initially 0, and set to 1 only when b_i is true. This is done through a set of constraints, one for each $r_i \in R$: $[(b_i \rightarrow bInt_i = 1) \wedge (\neg b_i \rightarrow bInt_i = 0)]$. Also, for each $r_i \in R$, we change the sub-term $\vee_{w'} Coupled_{r_i,w'}$ to $(ts_r < ts_f) \Rightarrow (b_i \Rightarrow \vee_{w'} Coupled_{r_i,w'})$ in DV, forcing the data-validity constraint for read r_i to hold when b_i is true. Note that with these changes, we require a different theory, that is *Linear Arithmetic* in the SMT solver to solve the constraints, compared to the *Difference Logic* which was used for our original set of constraints.

Initially, we set a threshold η to be $|R|$, the number of all read events. In each iteration, we assert the constraint $\sum_{1 \leq i \leq |R|} bInt_i = \eta$, which specifies the number (η) of data-validity constraints that should hold in that iteration. If no run can be predicted with the current threshold η (i.e. the constraint solver reports unsatisfiability), then η is decremented in each iteration, until the formula is satisfiable. This way, when a satisfying assignment is found, it is guaranteed to have the maximum number of reads that respect data-validity possible for predictable run.

Note that once $\eta < |R|$, the predicted run is not theoretically guaranteed to be a feasible run. However, in practice, when η is close to $|R|$ and a run is predicted, this run is usually feasible in the program.

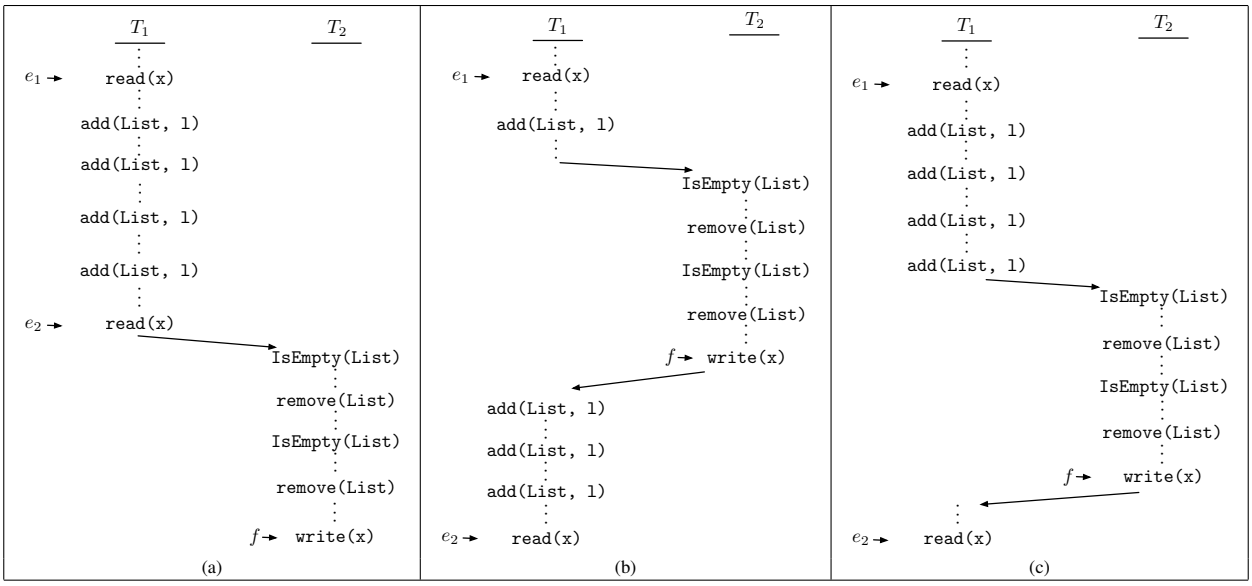


Figure 5.3: Example. A feasible vs. an infeasible atomicity violation prediction.

5.6 Precise Prediction of Other Concurrency Errors

Our proposed logic-based precise and relaxed prediction on statically pruned runs can be adapted to errors other than null-pointer dereferences as well. In order to evaluate our algorithms we implemented other two units in our tool. In particular we studied and implemented precise and relaxed algorithms for the prediction of atomicity-violating schedules.

5.6.1 Prediction of Atomicity-violating Schedules

We motivated in Chapter 4 the appeal of focusing on atomicity violations. In the following Subsections we motivate the need for precise prediction for this particular target. We present then the encoding for the precise prediction of atomicity violating runs and finally we present the encoding for predicting data-races.

5.6.1.1 Motivation for Precision

For each block of code marked atomic, even with a fixed interfering access, there are myriad ways of predicting schedules that violate atomicity; however, not all of these may actually be feasible in the program. It is computationally expensive to try all these executions to find a feasible one. Our goal is to find a guaranteed feasible one when possible.

Let us clarify this by an example. Consider the sample execution run R illustrated in Figure 5.3(a). Assume that the `read(x)` accesses in thread T_1 where intended to be atomic, and are required to see the same value of `x`, while the `write(x)` in thread T_2 changes the value of `x`. Also, assume that `List` is a global linked-list, and `add(List,`

`l)` inserts a new element with the value of local variable `l` at the beginning of the list, while each `remove(List)` actions in T_2 removes the first element of `List`. Let us also assume that each call to `remove` is preceded by a check by T_2 that calls `IsEmpty(List)` to check if the list is empty; if the list is empty, the path in T_2 will diverge from the run shown, and T_2 will not remove elements from the list, nor write to x. Let us also assume that each individual access to `List` and `x` is protected by synchronized methods that acquire the appropriate locks; hence there are no races in this program.

The prediction algorithms proposed in Chapter 4 generate a specific lock-valid execution and ignore data, would come up with the execution demonstrated in Figure 5.3(b). This execution, however, is *not feasible* since before the control reaches the `write(x)` event, the second call to `IsEmpty` will fail, and T_2 will diverge from the above path. Consequently, the `write(x)` event will never get executed, and the atomicity violation will not get scheduled.

The precise prediction algorithm we set in this Section ensures that in the new run every `IsEmpty` operation will see exactly the same version of List as it did in the original run. Therefore, if no exceptions were raised in the original run, no exceptions would be raised in the predicted run. The algorithm predicts the run in Figure 5.3(c). In this run, the violation occurs (and therefore the error will be discovered), and none of the `IsEmpty` calls will return true, and T_2 will continue to execute the `write x` event. The idea is that we guarantee that at every global variable read (`List` in this case), the predicted run and the original run agree, we can guarantee that the local computation that follows the read (dots in Figure 5.3), the values of local variables (that we do not observe), and the local computation in general, will remain the same as the original run. In our example, the value of local variable `l` may change during the local computation that we don't see in the figure, but since anything that would serve as an input to this local computation is unchanged, we can be assured that the result will be the same.

5.6.1.2 Encoding Precise Prediction of Atomicity Violations in Logic

Given a run ρ and an atomicity violation pattern (e_1, e_2, f) of events in ρ, we encode the set of feasible permutations of the events in the run in which atomicity is violated, with respect to the pattern, as a set of constraints and use SMT solvers to find such permutations. A target predicted run ρ' can be seen as a total ordering of the set of events E of the run ρ, where this ordering defines a run that is a precise predicted run of ρ and further has e_1 occurring before f, and e_2 not occurring at all in the run.

We introduce a new set of constraints: atomicity violation constraints (AV). The encoding consists now of the conjunction of these constraints:

$$\psi \equiv PO \wedge DV \wedge LV \wedge AV$$

. Atomicity Violation constraints: AV We express now the constraints that induce atomicity violations. Recall that this constraint forces that e_1 occur before f and that e_2 does not occur till f occurs. We cannot force the occurrence

of e_2. The constraint is simply: $AV = (ts_{e_1} < ts_f)$

The constraint that e_2 should not occur before f will be implicitly satisfied as we will not include e_2 in the set of scheduled events. The fact that *some* subset of events between e_1 and e_2 are scheduled is captured in the next Subsection.

Scheduling a subset of events from e_1 to e_2: We let the constraint solver decide which of the events between e_1 and e_2 will be scheduled. Intuitively, we only want that e_1 to be scheduled before f, and e_2 not occur in the interim. The events between e_1 and e_2 can however *help* in scheduling up till f; for instance, a write event in this range can help satisfy the constraint for a read event before f. But events in this range could also be hard to schedule; for example, a read event in this range may not have a corresponding write event. By allowing the constraint solver to choose a subset of events between e_1 and e_2, we give full rein to the prediction algorithm to find runs.

The events scheduled in this range *after* f do not entail the lock-validity and data-validity constraints. More precisely, for every event e that occurs after f, we drop the corresponding read constraint on it, i.e. we replace the condition $Coupled_{e,e'}$ in the formula DV to $(ts_e < ts_f \Rightarrow Coupled_{e,e'})$. Similarly, we drop the lock constraints on events occurring after f (this relaxation is more involved but straightforward).

The constraint generated above accurately captures the problem of precise prediction of violations:

Given a run ρ and events e_1, e_2, and f in ρ, the set of runs that satisfy the formula described above are the exact set of runs that correspond to the precisely predicted runs from ρ that violate atomicity.

The constraints generated fall within the class of quantifier-free difference logic constraints that are solvable in polynomial time, and which SMT solvers efficiently solve in practice. From Theorem 5.1.2, it follows that any of the runs predicted by using a logical constraint solver on the above formula is *necessarily* feasible in the program.

5.6.2 Data-races Prediction

A data-race occurs when two different threads in a given program can simultaneously access a shared variable, with at least one of the accesses being a write operation. Data-races often lead to unpredictable behavior of programs and are usually symptomatic of error, especially on programs such as those in our benchmark. In order to predict data-races, we first identify a set of access pairs $\alpha = (e, f)$ where e and f are accesses to the same shared variable in different threads and at least one of them is a write access. Using the lock-set and acquisition histories analysis, we prune the pairs for which there is no lock-valid run predictable. The set of constraints generated for data-races remains as described in Section 5.3, except that instead of null-reads constraint we require $(ts_e == ts_f)$, i.e. e and f should be co-reachable. We report the results of the evaluation of this unit on our set of benchmarks in Section 7.4.3.

Chapter 6

Implementation

We have implemented our approach in a tool named PENELOPE. In this Chapter, we describe the key implementation details of PENELOPE whose structure is illustrated in Figure 6.1. It consists of three main components: a monitor, a run predictor, and a scheduler. In the following, we will explain each of these components in more details. Source along with scripts and documentations are available from web-based Github[1] repository.

6.1 Monitor

The monitor component has an instrumenter which uses the Bytecode Engineering Library (BCEL) [7] to (automatically) instrument every class file in bytecode so that a *call* to an event recorder is made after each *relevant* action is performed. These relevant actions include:

- field and static field accesses
- array accesses
- acquisition and releases of locks
- thread creations

[1] https://github.com/sorfrancesco/Penelope

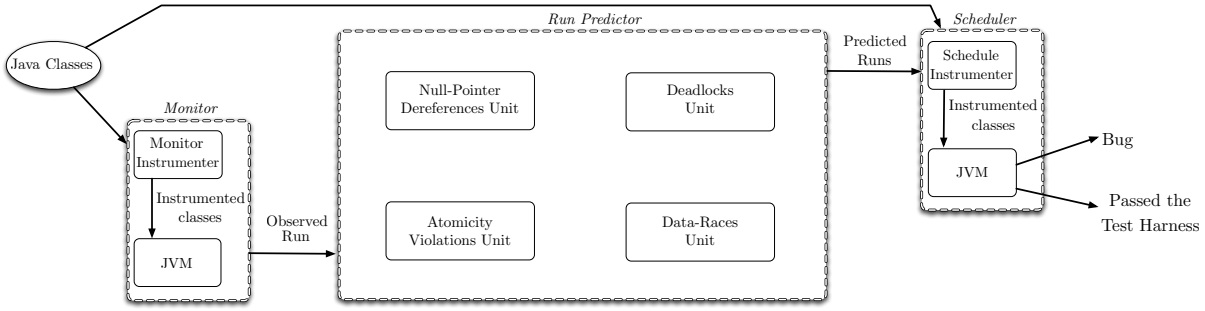


Figure 6.1: PENELOPE.

- thread joins

Notice that are excluded accesses to local variables. The instrumented classes are then used in the Java Virtual Machine (JVM) to execute the program and get an observed run.

For the purpose of generating the data-validity constraints, the values read/written by shared variable accesses are also recorded. For variables with primitive types (e.g. Boolean, integer, double, etc), we just use the values read/written. Objects and arrays are treated differently; the object *hash code* (by `System.identityHashCode()`) is used as the value every time an object or an array is accessed.

6.2 Run Predictor

The run predictor consists of several components that are combined depending on the prediction targeted and the precision demanded. It includes:

- cut-point generation
- segment generator
- constraint generator
- Z3 SMT solver
- run extractor/schedule generator

Cut-point generation. In this phase, all the possible *cut-points* are generated off-line: i.e. pairs of events (e, f), such that a schedule that reaches these points concurrently will expose the target. For the prediction of null-pointer dereferences, the cut-point generation (also referred as *null-WR* pairs extractor) generates a set of *null-WR* pairs from the observed run by the static lock analysis described in Section 5.2.3.

For the atomicity-violation prediction this phase generates all the pairs of events (e, f), such that a schedule that reaches these points concurrently will violate atomicity (see Section 4.2.4). Note that our algorithm computes only one representative violation for each pair of threads, each entity, each program location, each pair of events with a compatible set of lock-sets and acquisition histories, and each pattern of violation (W-R-W and A-W-A). Since *recurrent* locks (multiple acquisitions of the same lock by the same thread) are typical in Java, the tool is tuned to handle them by suppressing from the analysis the subsequent acquisitions of the same lock by the same thread. A simple automatic *escape analysis* unit (written as a Perl script) excludes from the execution all accesses to thread-local entities by replacing them with *skips* (nops), which enables faster prediction.

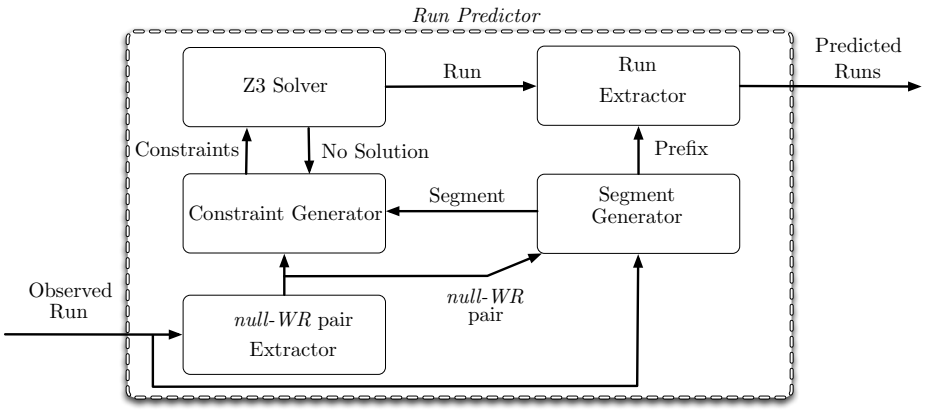


Figure 6.2: Run Predictor phase for null-pointer dereferences with the SMT solver based approach.

For deadlocks prediction the pair of events (e, f) are such that a schedule that reaches these points concurrently will deadlock (see Section 4.3.3.1).

Finally for data-races prediction the events e and f are accesses to the same shared variable in different threads and at least one of them is a write access.

Segment generator. The segment generator component, for each cut-point $\alpha = (e, f)$, isolates a part of run ρ that is *relevant* to α as described in Sections 4.4.2 and 5.4 for lightweight and precise prediction respectively, and passes it to the next component. The segment generated in this phase is fed to the constraint generator for the precise/relaxed prediction (see 5.3), and to the schedule generator for the lightweight prediction (see 4.4.1).

Constraint generator. For the precise and relaxed prediction given a cut-point and the relevant segment, the constraint generator produces a set of constraints, based on the algorithm presented in Section 5.3, and passes it to the Z3. Any model found by Z3 corresponds to a concurrent schedule.

Run extractor. The run extractor component is used in the precise and relaxed prediction. It generates a run based on the model returned by Z3. When Z3 cannot find a solution, the constraint generator iteratively weakens the constraints (see Section 5.5) and calls Z3 until a solution is found.

Schedule generator. The schedule generator, used in the lightweight prediction, synthesizes a schedule for each (predicted) *cut-point* using the algorithms described in Sections 4.4.1. The idea is that schedules are first synthesized up to the points where lock-sets are empty using the original observed schedule, and then the theoretical scheduling algorithm that adhere to the original execution (see Section 4.4.2) is used to execute the events of T_1 and T_2 to cause a bug (e.g. atomicity pattern violation, deadlock).

Also, we implemented the heuristics to reduce the number of context-switches by rearranging events that are not truly shared (using the *escape analysis*) as well as blocks of reads, to reduce the number of context-switches.

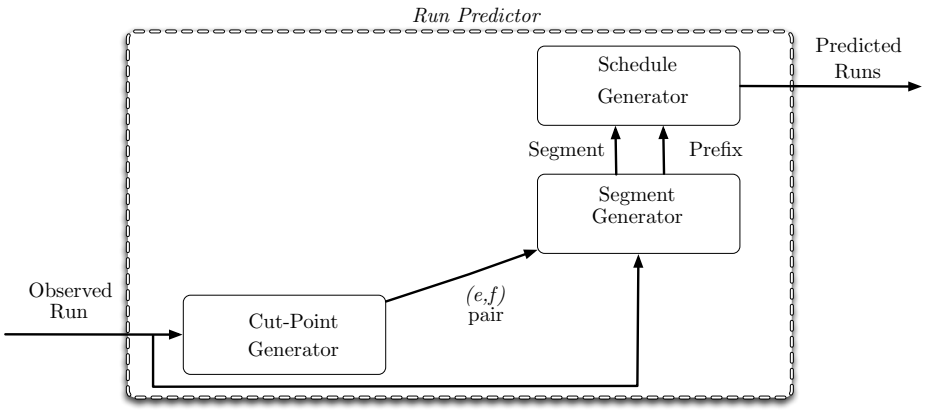


Figure 6.3: Run Predictor phase for Atomicity-violations with the lightweight approach.

In order to have a complete picture of the *Run Predictor* phase, in Figure 6.2 is reported the flow of the various components for the precise/relaxed prediction of null-pointer dereferences. In Figure 6.3 is reported the flow for the lightweight prediction of atomicity-violations.

6.3 Scheduler

The scheduler is implemented using BCEL [7] as well; we instrument the scheduling algorithm into the Java classes using bytecode transformations, so that the same events that were monitored, now interact with a global scheduler. The scheduler, at each point, looks at the predicted schedule, and directs the appropriate thread to perform a sequence of n steps. The threads stop at the first point with a relevant access, and wait for a signal from the scheduler to proceed, and only then, they execute the number of (observable) events they were asked to execute. After this, the threads communicate back to the scheduler, relinquishing the processor, and await further instructions.

The scheduler uses two vectors S and N to communicate with program threads. For each thread T_i, $N[i]$ will contain the number of steps that T_i should take next, and $S[i]$ contains a Boolean value that allows T_i progress when true. Only S is used to communicate with threads, and is protected by global locks (one for each cell). Thread T_i can safely access $N[i]$ without synchronization (as the synchronization in S is used to provide the synchronization for N as well). The thread $T[i]$ hence does a `wait()` on $S[i]$ (to become true), and the scheduler wakes up the thread when it is time for the thread to proceed. The scheduler then waits on $S[0]$ to become true. When the thread completes its designated number of steps (from $T[i]$), it sets $S[0]$ to true, notifies the scheduler, and then relinquishes control and waits on $S[i]$ to become true again. Once the execution reaches the point that the e and f event from the violation pattern are executed, the scheduler releases all threads to execute as they please. There is also a *timeout* mechanism that detects when the scheduler is trying to schedule an infeasible run.

Chapter 7

Experimental Evaluation

7.1 Configuration

Experiments were performed on an Apple MacBook with 2 Ghz Intel Core 2 Duo processors and 4GB of memory, running OS X 10.4.11 and Sun's Java HotSpot 32-bit Client VM 1.5.0. Source along with scripts and documentations are available from web-based Github[1] repository.

7.2 Benchmarks

The benchmarks used are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization. They include applications and data structures:

- `elevator`. A discrete events simulator for multiple lifts in a building developed at ETH Zurich [94].
- `TSP`. A multithreaded Java implementation of a traveling salesman problem solver for a given input map developed at ETH Zurich [94].
- `RayTracer`. A 3D raytracer (renders a frame of an arrangement of spheres from a given view point) from the JavaGrande multithreaded benchmarks [4].
- `Colt`. An open source library for high performance computing [10].
- `Pool` (releases 1.2, 1.3, 1.5). An object pooling API in the Apache Commons Project [8].
- `DBCP`. A Database Connection Pool in the Apache Commons suite[3].
- `Apache FtpServer`. A pure Java FTP server designed to be a complete and portable FTP server engine solution [11].
- `Hedc`. A Web crawler application[9].

[1] https://github.com/sorfrancesco/Penelope

- `Weblech`. A websites download tool[19].
- `Vector`, `Stack`, `HashSet`, `StringBuffer`, `ArrayList`, `PrintWriter`, `Logging` and `Hashmap` are standard library classes of Java [14].
- `StaticBucketMap`. A thread-safe implementation of the Java Map Interface from the Apache Commons Project [8].

7.3 Test Suites

The benchmarks were run against several test harnesses and input parameters. The tables in this Section provide all the relevant information about the conditions under which the tests were run. For `elevator`, and `TSP`, the input files were included in the benchmarks. For `elevator`, the number of threads was also specified in the input files, and there were no additional parameters to be provided by the user. The test harness for `TSP` includes an input file, a given number of threads, and a script would compare the minimum tour computed by the program against the minimum tour computed by a single thread execution. For `RayTracer`, all the data is already incorporated in the benchmark which comes in two sizes A, and B. The only parameter that the user specifies is the number of threads. `RayTracer` has a built-in validation test. For `colt`, the `dgemm` command was used which invokes a matrix multiplication routine on matrices of size 50×50. The test harness was designed to check the result from a multi-threaded execution against the result from a single threaded execution. For `Pool` we wrote test cases, for each test case two different methods were run concurrently on the same object pool. `DBCP` was designed in a similar manner. In the case of `FtpServer`, we wrote a test harness with a client and a server where the client logs in and requests a connection. There is no test harness for `Hedc` and we just ran the program. For `Weblech`, we just downloaded a website using the standard settings.

In `Vector`, we wrote test cases with two threads and two small vectors in which each thread executes exactly one method from class `Vector`. Test cases for `Stack`, `HashSet`, `StringBuffer`, `ArrayList`, `PrintWriter`, `Logging`, `HashMap` and `StaticBucketMap` were designed in a similar manner.

7.4 Evaluation

In this Section we show the result of the evaluation of PENELOPE on the different error targeted. For the sake of readability in the rest of the Section we will refer to the lightweight version with PENELOPE v1 (in which the predicted schedules are only guarantee to be lock-valid) and to the SMT logical solver version with PENELOPE v2.

		Monitoring						Prediction		Scheduling						
Application (LOC)	Execution Time	Threads	Entities	Locks	Execution Length	Execution Time	Context Switches	Number of Predicted Schedules	Execution Time	Patterns A	Patterns N	Patterns F	Context Switches	Average Time per feasible schedule	Total Time rescheduling	Errors
Stack (1.4K)	0.21s	4	12	2	136	0.24s	6	2	0.01s	0	0	2	5	0.32s	1.06s	1
	0.48s	4	12	2	209	0.58s	6	1	0.01s	0	0	1	6	0.56s	1.35s	1
	0.29s	4	12	2	231	0.28s	5	3	0.01s	0	1	2	5	0.35s	1.77s	1
	0.17s	4	12	2	265	0.27s	6	3	0.01s	0	0	3	6	0.18s	1.20s	1
	0.18s	4	12	2	127	0.18s	4	2	0.01s	0	0	2	4	0.19s	1.04s	1
Vector (1.3K)	0.22s	4	12	2	142	0.29s	5	2	0.01s	0	0	2	5	0.19s	1.10s	1
	0.20s	4	12	2	231	0.20s	5	4	0.01s	0	1	3	5	0.33s	1.61s	1
	0.18s	4	12	2	248	0.20s	5	4	0.01s	0	0	4	5	0.23s	1.57s	1
	0.27s	4	12	2	231	0.27s	5	3	0.01s	0	0	3	5	0.21s	1.27s	1
	0.33s	4	12	2	159	0.35s	3	1	0.01s	0	0	1	3	0.35s	1.27s	1
Colt (29K)	0.27s	3	12K	0	286K	0.42s	358	0	1.7s	0	0	0	297	-	-	0
Elevator (566)	16.37s	3	65	8	26K	18.94s	899	167	5.24s	0	164	3	595	16.78s	30m42s	0
	16.77s	5	113	8	54K	16.26s	1428	63	3.87s	0	59	4	1262	17.12s	11m44s	0
	16.37s	5	457	50	329K	16.93s	50K	699	6m48s	42	651	6	39K	17.13s	2h52m	0
TSP (794)	0.19s	2	588	2	32M	3m41s	72K	83	1m36s	29	45	9	60K	15.85s	50m01s	0
	0.15s	4	652	2	14M	1m58s	22K	168	29.98s	87	70	11	18K	16.16s	23m41s	0
Pool 1.2 (5.8K)	0.19s	4	28	1	98	0.19s	11	1	0.01s	0	0	1	4	0.17s	0.18s	1
	0.20s	4	29	1	267	0.24s	25	27	0.01s	0	3	24	19	0.24s	8.20s	1
	0.12s	4	15	1	104	0.17s	5	7	0.01s	0	0	7	5	0.20s	1.48s	1
	0.12s	4	14	1	251	0.18s	20	145	0.01s	50	46	36	49	0.20s	20.76s	1
Pool 1.3 (7K)	0.25s	4	30	1	100	0.21s	3	0	0.01s	0	0	0	5	-	-	0
	0.18s	4	31	1	265	0.25s	14	19	0.02s	6	13	0	14	-	5.78s	0
	0.12s	4	15	1	112	0.17s	5	6	0.01s	0	6	0	5	-	1.06s	0
	0.12s	4	18	1	250	0.88s	24	99	0.01s	52	47	0	22	-	11.31s	0
RayTracer (1.5K)	3.38s	10	80	10	560	3.57s	54	90	0.01s	0	0	90	48	3.57s	5m52s	1
	3.94s	20	150	20	1.5K	4.03s	106	380	0.03s	0	0	380	93	4.14s	26m20s	1
	6.52s	30	220	30	2.9K	6.08s	150	870	0.09s	0	0	870	99	6.75s	1h38m	1
	36.50s	10	80	10	560	36.62s	59	90	0.01s	0	0	90	44	38.46s	58m16s	1
Apache FtpServer (22K)	1m02s	5	67	4	412	1m02s	9	109	0.02s	5	83	31	9	1m03s	2h16m	5

Table 7.1: Experimental Results. A, N and F (of Patterns) indicate number of schedules that are, respectively, "Already appeared in observed execution", "Not feasible" and "Feasible".

7.4.1 Atomicity-Violations

Table 7.1 demonstrates the results of the evaluation of PENELOPE v1 on the benchmarks while targeting atomicity-violations [91]. The table provides information about all three phases: monitoring, prediction, and scheduling. For the monitoring phase, the number of threads, entities (variables), locks, and the length of execution is reported, as well as how long the execution takes, and how many context switches exist in the observed run. For the prediction phase, we report how many violations were discovered (total over all 5 patterns), and how long the prediction phase takes. In the scheduling part, we report the number of violation patters (out of the total reported in the prediction column) existed in the original run observed (A), the number of violations for which the schedules were not feasible (N), and the number of violations which appeared in a successful alternative schedule generated by PENELOPE v1 (F). We also report how many context switches (on average) there are in the alternative feasible schedules, what is the average time per feasible schedule (a reasonable indication of overhead), and finally and most importantly, how many *real* errors were found.

PENELOPE v1 finds several bugs in the benchmarks. The bug in `raytracer` is caused by an atomicity violation involving the field `JGFRayTracerBench.checksum1` due to wrong synchronization. The error in `Pool 1.2` is caused by an atomicity violation on the variable `_factory` in methods `borrowObject`, `returnObject`, ran in parallel with method `close`, and in methods `addObject` and `borrowObject` ran in parallel with method `setFactory`. Note that these are 4 different errors, and they all manifested as exceptions during alternative feasible schedules exercised by PENELOPE v1.

All bugs in `Vector` and `Stack` are the result of an atomicity violation that causes the size of a parameter collection to go stale in the middle of an operation (such as `addAll`) that is using the parameter collection as a source of information, while only the destination vector/stack is synchronized properly. There were 5 discovered errors for `FtpSever` (each giving a different exception), which correspond to variables `m_currConnections`, `m_writer`, `m_name`, `m_currLogins`, `m_request` all accessed in method `RequestHandler.run` of the server while the timer thread interrupts by closing the connection as a result of a timeout.

PENELOPE v1 predicts several atomicity violating schedules in `tsp` and `elevator`, but they all pass the test harness, and in fact are not errors (the violation of atomicity was intended and correct).

From the experiments it is clear that the number of predicted schedules is small. In fact, a tiny fraction of all possible executions. This is true even compared to the number of all runs limited to just two context switches (or preemptions), as CHESS [70] would do. For instance, in `elevator` (data3), there are close to 50,000 points (releases of locks) in 4 threads where preemptions can happen, giving around 15 billion possible schedules involving just 2 preemptions!

PENELOPE v1 is effective in finding bugs moreover it add a reasonable overhead. We ran the programs under the test harness several times, and did not find any of the reported bugs in any of these benchmarks by merely running tests randomly. It is clear that a more focused approach is absolutely necessary in finding errors on these benchmarks. Despite its small selection of schedules to test, PENELOPE v1 was able to identify bugs in these programs.

The runtime overhead in scheduling the alternate executions is not prohibitively high, and is in fact very minimal in most examples. This is despite the large number of context-switches that are being exercised (284K context-switches in elevator-data3). PENELOPE v1 finds bugs under complex scenarios. Note that the number of context-switches scheduled in the predicted executions are very high. We believe that this allows PENELOPE to dig deep into the search space of the runs. Tools like CHESS execute all runs with a few context-switches and offer a complementary search strategy [70].

If a bug is reported by PENELOPE v1, it is a real bug (i.e. an execution that violates the test harness). A significant amount of violations found did not correspond to real bugs, and are not reported as bugs. This is in contrast to similar tools based on atomicity checking by Wang and Stoller [98], Farzan et al. [40], and the tools SIDE-

		Monitoring					Prediction	Feasible Schedules		Total Time		Errors
Application (LOC)	Input	Threads	Entities	Locks	Execution Length	Execution Time	Number of Access Patterns	PENELOPE v1	PENELOPE v2	PENELOPE v1	PENELOPE v2	
Stack (1.4K)	ST1	4	24	2	319	<1s	5	5	**5**	2s	3s	1
	ST2	4	24	2	354	<1s	18	18	**18**	6s	7s	1
Vector (1.3K)	VT2	4	24	2	350	<1s	23	23	**23**	4s	8s	1
	VT3	4	24	2	353	<1s	3	2	**3**	<1s	2s	1
	VTt4	4	24	2	319	<1s	5	5	**5**	2s	3s	1
Elevator (566)	data2	5	149	8	27K	7s	4	4	**4**	24s	25s	0
	data3	5	503	50	63K	19s	43	41	**42**	13m5s	18m28s	0
TSP (749)	Map10-2	3	15K	2	350K	<1s	65	53	**54**	3m21s	25m49s	0
Pool 1.2 (5.8K)	PT1	4	29	1	267	<1s	26	25	**25**	5s	15s	1
	PT2	4	15	2	177	<1s	31	30	**31**	6s	21s	1
	PT3	4	18	2	356	<1s	374	275	**365**	1m09s	5m32s	1
Pool 1.3 (7K)	PT1	4	31	1	270	<1s	19	17	**18**	4s	11s	1
	PT2	4	15	2	204	<1s	22	17	**22**	3s	13s	1
	PT3	4	18	2	422	<1s	417	211	**381**	1m23s	6m04s	1
Static-BucketMap (750)	SBMTest	4	123	19	892	<1s	1	1	**1**	<1s	<1s	1
RayTracer (1.5K)	A-10	10	106	10	648	3s	90	90	**90**	4m02s	4m49s	1
	A-20	20	196	20	1.7K	4s	380	380	**380**	17m14s	23m55s	1
	B-10	10	106	10	648	36s	90	90	**90**	34m18s	35m56s	1
Apache FtpServer (22K)	lgn_spt	5	112	4	578	1m03s	33	7	**33**	34m15s	34m26s	3+1

Table 7.2: Experimental Results. Comparison PENELOPE v1 and PENELOPE v2 on prediction of atomicity violations.

TRACK [106], ATOMFUZZER [74] and VELODROME [44]. VELODROME in fact reports atomicity violations for benchmarks `elevator`, `tsp`, and `colt`, though they do *not* correspond to bugs.

7.4.1.1 Lightweight Vs. SMT Solver

In Table 7.2 we compare PENELOPE v1 (in which the predicted schedules are only guarantee to be lock-valid) with PENELOPE v2.

Table 7.2 provides information about the benchmarks as well as information about all three phases: monitoring, prediction, and scheduling. Notice that even if monitoring information and time information (for PENELOPE v1) have been presented already in Table 7.1 we report them again here for two reasons. (1) the observed run utilized in this analysis is different than the one used in Table 7.1, (2) the version of PENELOPE v1 used in this experiment is a more mature implementation of the one used in Table 7.1 (in average 35% faster).

For the prediction phase, we report the number of access patterns in the observed run. In the schedule generation part, we report the number of feasible schedules generated by PENELOPE v1 and PENELOPE v2, followed by total time taken for predicting and rescheduling generated schedules. We also provide information about the number of *real* errors that were found using Penelope v2.

From the Table 7.2 it is clear that PENELOPE v2 produces more feasible schedules. We can see that for some programs like `FtpServer`, `Pool 1.2`, and `Pool 1.3` the number of feasible schedules generated by PENELOPE v2 is considerably greater than the number of feasible schedules generated by PENELOPE v1. This is because of

Application	Input	Number of Precisely Predicted Schedules	Number of Feasible Schedules with Relaxed Data-Validity	Number of Feasible Schedules in PENELOPE v2
`Stack`	StackTest1	2	5	5
	StackTest2	18	18	18
`Vector`	VectorTest2	23	23	23
	VectorTest3	3	3	3
	VectorTest4	2	5	5
`Elevator`	Data2	4	4	4
	Data3	42	43	43
`TSP`	Map10-2	26	54	54
`Pool 1.2`	PoolTest1	24	25	25
	PoolTest2	4	31	31
	PoolTest3	280	334	365
`Pool 1.3`	PoolTest1	16	18	18
	PoolTest2	8	22	22
	PoolTest3	195	372	381
`StaticBucketMap`	SBMapTest	1	1	1
`RayTracer`	SizeA-10	45	90	90
	SizeA-20	190	380	380
	SizeB-10	45	90	90
`FtpServer`	lgn_script	29	33	33

Table 7.3: Feasibility rates for schedule generation algorithms.

the fact that the schedules generated by PENELOPE v1 are only lock-valid and there is no guarantee for them to be feasible when threads communicate using shared variables. On the other hand, PENELOPE v2 first tries to find a both lock-valid and data-valid schedule and if it fails then relaxes data-validity.

PENELOPE v2 finds a new bug that PENELOPE v1 does not. In `FtpServer`, there are four different bugs. One of the bugs, found by PENELOPE v2, could not be find in PENELOPE v1. There was also another bug which was found by PENELOPE v1 but PENELOPE v2 did not discover it, but the discovery of that bug is a lucky accident and is not related to an atomicity violation pattern. Please notice that the observed run used in this analysis is different than the one used in Table 7.2 (5 errors found in that case).

Table 7.3 contains information about the number of precisely predictable schedules and the number of feasible schedules found after weakening data-validity in PENELOPE v2. We can see that the *relaxed data-validity* allows PENELOPE v2 to find more feasible schedules. For, `TSP`, `Pool`, and `Raytracer`, considerably more feasible patterns can be found by relaxing data-validity. In practice, exempting a few reads from being matched greatly improves the flexibility of the constraint solver in predicting runs. In most cases, relaxing 1 or 2 reads was enough to find a solution.

7.4.2 Null-pointer Dereferences

Table 7.4 illustrates the experimental results for *null-pointer dereference prediction* using PENELOPE v2. Information are provided about all the three phases of monitoring, run prediction, and scheduling.

			Monitoring					Prediction			Scheduling				
Application (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Schedulable Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
HashSet (1.3K)	HT1	<1s	4	76	1	432	<1s	7	7	-	7	<1s	3.2s	**1**	**0**
	HT2	<1s	4	54	1	295	<1s	0	-	-	-	-	<1s	**0**	**0**
Stack (1.4K)	ST1	<1s	4	29	2	205	<1s	11	6	5	11	<1s	5.5s	**2**	**0**
	ST2	<1s	4	24	2	251	<1s	16	11	5	15	<1s	10.9s	**1**	**0**
	ST3	<1s	4	24	2	248	<1s	17	12	5	17	<1s	10.3s	**1**	**0**
	ST4	<1s	4	29	2	515	<1s	30	0	30	30	1.8s	53.2s	**0**	**1***
	ST5	<1s	4	29	2	509	<1s	85	1	84	83	2.0s	2m51s	**0**	**1***
StringBuffer (1.4K)	SBT	<1s	3	16	3	80	<1s	2	2	-	2	<1s	1.3s	**1**$^{+}$	**0**
Vector (1.3K)	VT1	<1s	4	44	2	370	<1s	21	11	10	21	<1s	14.3s	**2**	**0**
	VT2	<1s	4	34	2	536	<1s	31	21	10	31	1.1s	33.0s	**1**	**0**
	VT3	<1s	4	34	2	443	<1s	32	22	10	32	<1s	22.1s	**1**	**0**
	VT4	<1s	4	29	2	517	<1s	30	0	30	30	2s	59.4s	**0**	**1***
	VT5	<1s	4	29	2	505	<1s	85	1	84	82	2s	2m57s	**0**	**1***
Elevator (566)	Data	7.3s	3	116	8	14K	7.4s	0	-	-	-	-	7.9s	**0**	**0**
	Data2	7.3s	5	168	8	30K	7.4s	0	-	-	-	-	8.9s	**0**	**0**
	Data3	19.2s	5	723	50	150K	19.0s	0	-	-	-	-	58.5s	**0**	**0**
Pool 1.2 (5.8K)	PT1	<1s	4	28	1	98	<1s	3	2	1	3	<1s	1.6s	**2**	**0**
	PT2	<1s	4	29	1	267	<1s	3	0	0	-	-	8.8s	**0**	**0**
	PT3	<1s	4	20	3	180	<1s	26	0	23	16	1.2s	27.0s	**0**	**3**
	PT4	<1s	4	24	3	360	<1s	32	2	21	15	2.5s	57.8s	**0**	**1**
Pool 1.3 (7K)	PT1	<1s	4	30	1	100	<1s	3	0	3	3	<1s	2.6s	**0**	**0**
	PT2	<1s	4	31	1	271	<1s	3	0	0	-	-	9.8s	**0**	**0**
	PT3	<1s	4	20	3	204	<1s	35	0	30	19	1.4s	42.9s	**0**	**0**
	PT4	<1s	4	23	3	422	<1s	62	1	48	29	2.2s	1m49s	**0**	**1**
Pool 1.5 (7.2K)	PT1	<1s	4	33	2	124	<1s	2	0	1	1	1.5s	1.5s	**0**	**0**
	PT2	<1s	4	34	2	306	<1s	5	0	1	0	10.5s	10.5s	**0**	**0**
	PT3	<1s	4	15	2	108	<1s	3	0	0	-	-	4.1s	**0**	**0**
	PT4	<1s	4	18	2	242	<1s	18	1	7	8	3.4s	27.4s	**0**	**1**
SBucketMap (750)	SMT	<1s	4	123	19	892	<1s	2	2	-	2	<1s	1.3s	**1**	**0**
RayTracer (1.5K)	A-10	5.0s	10	106	10	648	5.0s	9	9	-	9	5.6s	50.5s	**1***	**0**
	A-20	3.6s	20	196	20	1.7K	4.4s	19	19	-	19	6.7s	2m15s	**1***	**0**
	B-10	42.4s	10	106	10	648	42.5s	9	9	-	9	42.7s	6m24s	**1***	**0**
Hedc (30K)	Std	1.7s	7	110	6	602	1.74s	18	9	1	10	11.7s	1m57s	**1**	**0**
Weblech v.0.0.3 (35K)	Std	4.9s	3	153	3	1.6K	4.92s	55	10	29	30	16.26s	10m34s	**1**	**1**$^{@}$
Apache FtpServer (22K)	LGN	1m2s	4	112	4	582	60s	116	78	32	65	1m13s	2h14m46s	**9**	**3**
											Total Number of Errors			**27**	**14**

Table 7.4: Experimental Results for predicting null-reads. Errors tagged with * represent test harness failures. Errors tagged with $^{+}$ represent array-out-of-bound exceptions. Errors tagged with $^{@}$ represent unexpected behaviors. All other errors are null-pointer dereference exceptions.

In the monitoring phase, the number of threads, shared variables, locks, the number of potential interleaving points (i.e. number of global events), and the time taken for monitoring are reported.

In the prediction phase, we extend the set of *null-WR* pairs to also include access pairs (e, f) where e is writing zero to a shared variable and f is reading a non-zero value from the same variable in the original run. The intuition behind considering those access pairs is that some of the errors in the program (e.g. division by zero, or control flow bugs) can be exposed by forcing a read event to read a zero value. For the prediction phase, we report the number of *null-*

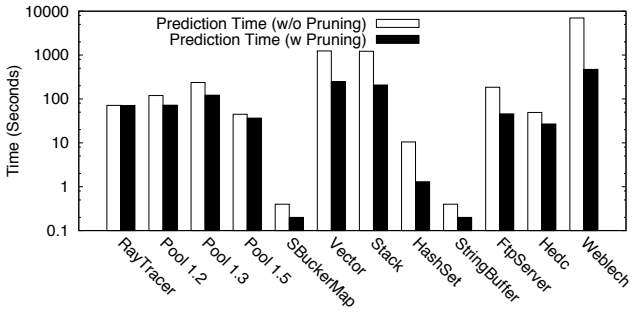


Figure 7.1: Prediction times with/without pruning in log scale.

WR pairs in the observed run, the number of precisely predicted runs, and the additional number of predicted runs after relaxing the data-validity constraints (when there is no precisely predicted run for a null read-write pair). Note that no predictable run can be found for some *null-WR* pairs even after weakening data-validity; this is due to the fact that lock analysis (described in section 5.2.3) is only a heuristic, and for some α pairs, a lock-valid run may not exist. However, the SMT solver can recognize and prune such *null-WR* pairs . In the scheduling phase, we report the total number of schedulable predictions among the predicted ones. Finally, we report the average time for prediction and rescheduling of each run, the total time taken to complete the tests (for on all phases on all predicted executions), and also the number of errors found using the precise and relaxed predicted runs.

Errors Found. In almost all the cases, the errors manifested in the form of raised exceptions during the execution. In `Weblech`, in addition to a null-pointer dereference, an unwanted behavior occurred (the user is asked to push a stop button even after the website is downloaded completely, resulting in non-termination!). `RayTracer` has a built-in validation test which was failed in some of the predicted runs. For some of the test cases of `Vector` and `Stack` the output produced was not the one expected. We report the errors found in two categories; those that were found through the precise prediction algorithm, and those that were found after weakening data-validity constraints (relaxation).

7.4.2.1 The Effect of Pruning

Figure 7.1 illustrates the substantial impact of our pruning algorithm in reducing prediction time. We present prediction times with and without using the pruning algorithm. Note that the histogram is on a logarithmic scale. For example, in the case of `Weblech`, the prediction algorithm is about 16 times faster with pruning. Furthermore, all errors found without the pruning were found on the pruned runs, showing that the pruning did not affect the quality of error-finding on our benchmarks.

Application	Input	Execution Time	Number of Racing Pairs	Number of Precisely Predicted Runs	Total Number of Predicted Runs	Total Number of Schedulable Predictions	Average Time per Predicted Run	Total Distinct Racing Configurations	Total Distinct Racing Variables
HashSet (1.3K)	HSTest1	<1s	20	20	20	20	<1s	6	3
	HSTest2	<1s	3	0	3	3	<1s	3	2
Stack (1.4K)	StackTest1	<1s	0	-	-	-	-	-	-
	StackTest2	<1s	0	-	-	-	-	-	-
	StackTest3	<1s	0	-	-	-	-	-	-
	StackTest4	<1s	5	1	5	5	1.14s	2	2
	StackTest5	<1s	0	-	-	-	-	-	-
Vector (1.3K)	VectorTest1	<1s	0	-	-	-	-	-	-
	VectorTest2	<1s	0	-	-	-	-	-	-
	VectorTest3	<1s	0	-	-	-	-	-	-
	VectorTest4	<1s	5	1	5	5	1.16s	2	2
	VectorTest5	<1s	0	-	-	-	-	-	-
Elevator (566)	Data	7.3s	0	-	-	-	-	-	-
	Data2	7.3s	0	-	-	-	-	-	-
	Data3	19.2s	0	-	-	-	-	-	-
Pool 1.2 (5.8K)	PoolTest1	<1s	2	2	2	2	<1s	2	1
	PoolTest2	<1s	7	2	7	7	<1s	2	2
	PoolTest3	<1s	5	0	5	5	<1s	0	0
	PoolTest4	<1s	7	1	7	4	<1s	1	1
Pool 1.3 (7K)	PoolTest1	<1s	0	-	-	-	-	-	-
	PoolTest2	<1s	0	-	-	-	-	-	-
	PoolTest3	<1s	0	-	-	-	-	-	-
	PoolTest4	<1s	0	-	-	-	-	-	-
Pool 1.5 (7.2K)	PoolTest1	<1s	1	1	1	1	1,15s	1	1
	PoolTest2	<1s	9	6	9	8	<1s	8	3
	PoolTest3	<1s	1	1	1	1	<1s	1	1
	PoolTest4	<1s	0	-	-	-	-	-	-
SBucketMap (750)	SBMapTest	<1s	1	1	1	1	<1s	1	1
RayTracer (1.5K)	SizeA-10	5.0s	145	46	136	135	4.27s	1	1
	SizeA-20	3.6s	590	106	570	569	4.52s	1	2
	SizeB-10	42.4s	145	46	136	135	44.11s	1	1
Hedc (30K)	std.prop	1.71s	30	26	30	26	14s	5	4
Weblech-0.0.3 (35K)	spider.prop	4.91s	31	6	31	27	21.20s	9	3
Apache FtpServer (22K)	lgn_script	60s	93	52	91	40	57s	11	7

Table 7.5: Experimental Results. Racing Configurations are (Function_name:PC_1, Function_name:PC_2).

7.4.3 Data-races

Table 7.5 presents the results of data-race prediction using PENELOPE v2 on our benchmarks using the same observed runs as in the null-reads prediction. For each benchmark we report the total number of data-races found; these are all distinct races identified by the code location of the racy access. We also report the number of distinct variables involved in data-races.

7.4.4 Deadlocks

Table 7.6 provides information about the benchmarks for *deadlocks* as well as information about all three phases: monitoring, run prediction, and scheduling. For the monitoring phase, the number of threads, the number of synchronization events, and the total number of observed events are reported, as well as the monitoring time. For the

			Monitoring				Prediction	Scheduling		
Application (LOC)	Input	Base Time	Number of Threads	Number of Syncs Events	Number of Observed Events	Time to Monitor	Number of Predicted Runs	Number of Schedulable Predictions	Total Time	Deadlocks Found
ArrayList (1.6K)	AL	$< 1s$	4	60	783	$< 1s$	1	1	1.2s	**1**
HashMap (1.3K)	HM1	$< 1s$	4	28	138	$< 1s$	1	1	1.3s	**1**
Java Logging (43K)	$id\_6487638$	<1s	5	228	1.4K	< 1s	2	2	2.3s	**1**
PrintWriter (1.2K)	PW	$< 1s$	4	14	80	$< 1s$	1	1	1.2s	**1**
Stack (1.4K)	ST1	<1s	4	112	527	<1s	1	1	1.1s	**1**
	ST2	<1s	4	14	69	<1s	1	1	1.3s	**1**
StringBuffer (1.4K)	SBT1	<1s	3	18	82	<1s	1	1	1.2s	**1**
	SBT2	<1s	4	16	75	<1s	1	1	1.1s	**1**
Vector (1.3K)	VT1	<1s	4	108	525	<1s	1	1	1.0s	**1**
	VT2	<1s	4	12	63	<1s	1	1	1.1s	**1**
Elevator (566)	Data	7.3s	3	6.6K	14K	7.4s	0	-	7.6s	**0**
	Data2	7.3s	5	22K	30K	7.4s	0	-	7.5s	**0**
	Data3	19.2s	5	138K	150K	19.4s	0	-	19.9s	**0**
DBCP 1.2.1 (168K)	DBCP-44	1.6s	4	216	1.5K	1.8s	2	1	4.1s	**1**
RayTracer (1.5K)	A-10	5.0s	10	20	648	5.0s	0	-	5.2s	**0**
	A-20	3.6s	20	40	1.7K	4.4s	0	-	4.1s	**0**
	B-10	42.4s	10	20	648	42.5s	0	-	43.1s	**0**
Hedc (30K)	Std	1.71s	7	198	774	1.74s	0	-	1.73s	**0**
Weblech v.0.0.3 (35K)	Std	4.91s	3	114	1.6K	4.92s	0	-	4.95s	**0**
Apache FtpServer (22K)	LGN	60s	4	20	582	1m1s	0	-	1m2s	**0**

Table 7.6: Experimental Results for deadlocks prediction using PENELOPE v1.

prediction phase, we report the number of potential deadlocks found. In the scheduling part, we report the total number of schedulable predictions. Finally, we present the total time for the test and the number of deadlocks found.

PENELOPE v1 found all previously known deadlocks in the benchmarks analyzed. The test cases for Java Collections in particular are artificial; these are not real bugs in the JDK Collections but could become errors if the clients use their API erroneously. We used such test cases to validate our tool considering that they were also used in previous works [57, 56, 71, 101].

The deadlock found in package $java.util.logging$ is reported at $http://bugs.sun.com/view\_bug.do?bug\_id = 6487638$. The deadlock occurs when the synchronized instance method $LogManager.addLogger()$ and the synchronized static method $Logger.getLogger()$ are invoked concurrently. In particular, one thread locks the $LogManager$ object, calls the $LogManager.addLogger()$ which calls $Logger.getLogger()$ which is waiting to lock the $Logger$ Class object. Another thread locks the $Logger$ Class object, calls $Logger.getLogger()$ which calls $LogManager.getLogger()$ which is waiting to lock the $LogManager$ object.

The deadlock found in DBCP is reported at $https://issues.apache.org/jira/browse/DBCP-44$. Such deadlock occurs when a thread from the function $GenericObjectPool.addObject()$ invokes the synchronized method

$PoolableConnectionFactory.makeObject()$, meanwhile concurrently another thread invokes the static synchronized $DriverManager.getConnection()$ method. $makeObject()$ waits for the lock on $DriverManager$ to be released, whereas $DriverManager$ waits for the lock on the $PoolableConnectionFactory$ instance to be released. The deadlocks for the Java Collections Framework happen when multiple threads sharing objects call concurrently methods of the class as described in the Paragraph 4.3.1 (we refer the reader to [101] for more details).

Chapter 8

Improving Testing and Diagnosis of Scalable Distributed Systems through Perturbation-based Learning

8.1 Introduction

Modern scalable distributed systems (SDS) are designed to provide 24/7 elastic seamless services. These systems can scale horizontally with addition of cheap commodity servers to support increasing load requests in a cost effective manner. As the commodity hardware components have higher failure rates than enterprise systems, these systems are specifically designed to be robust and resilient to various environmental anomalies such as node failures, loss of messages, network delays, and data corruptions. The software layers managing these anomalies are quite complex, and are carefully engineered to provide subtle trade-offs between consistency and availability. Testing such complex software in a large-scale distributed setting is not only costly but also very challenging. Often severe system-level design defects stay hidden even after deployment and can derail the entire system, possibly resulting in loss of revenue or customer satisfaction.

Motivation. In a recent work [53], a cost effective perturbation-based testing framework called SETSUDŌ was proposed to test the robustness of distributed systems in a pseudo-distributed setting (small-scale). A perturbation is the act of inducing controlled changes to the execution of the SUT, e.g. a forced invocation of an I/O exception handler. The basic idea is create stress tests that model real-time environmental perturbation, and control the application of the stresses on the running system at chosen system execution points. After the system is subjected to a sequence of perturbation (e.g. leader node failure, network connection failure), availability of the system is monitored. Availability failures are indicative of the system defects, which can be potentially replayed using the applied perturbation sequence [64].

In particular, a simplified and an effective perturbation testing has been proposed, where a single perturbation is applied to a system-under-test (SUT) at a time[1]. Before next perturbation is applied, the test engine waits until the effect of the applied perturbation is 'detected' (by anomaly detection code), and 'absorbed' (by the anomaly management code through corresponding actions). Note, after applying each perturbation, the system reacts, i.e., it

[1]The idea of single perturbation is similar to single stuck-at fault model used in post-manufacturing testing of digital circuits. This simple fault model assumes one line or node in the digital circuit is stuck at logic high or logic low. On one hand it is fairly easy to create a test for such a fault model, and on the other hand a test developed for such a fault often finds a large number of other stuck-at faults.

goes through internal changes such as a new leader election process. Corresponding to these internal changes, we observe a flurry of messages in log outputs, which stop after some time. In SETSUDŌ, there is a wait of 1-2 minutes after the last update of log message by any node, at which point it is inferred that SUT has completely reacted to the perturbation applied (i.e., reached a steady state). In general, such a conservative approach (i.e., waiting to ensure there is no further log update) is quite time consuming, especially, when one needs to apply millions of perturbations. One can reduce the wait time by knowing exactly when the system has absorbed the changes. This requires heavy-weight instrumentation which will incur performance overhead, resulting in increased testing time instead of reducing it. Informally, we refer this issue as *qualitative* a testing problem, i.e., *can we reduce the testing time without incurring performance overhead?*

Defect diagnosis of a distributed system is also quite challenging. Generally the effects of environmental perturbations to the running system (i.e., reactions of the system to the perturbations) are logged. Given a sequence of recorded system states prior to the observed system defects, it may be useful to know what the environmental perturbations were that triggered the system failure. Knowing such triggers often helps to recreate the defect. Informally, we refer this problem as a diagnosis problem, i.e., *can we infer a possible perturbation sequence from a given sequence of system states?*

Basic Idea. We built our approach over the SETSUDŌ testing framework. In a pseudo-distributed setting (i.e., distributed but not at a very large scale), we apply perturbation sequences corresponding to environmental anomalies, and record system states in response to each perturbation. On these recorded observations, we use supervised learning based on decision tree classifiers to model the system behaviors in response to perturbation. Using the learned system model, we predict the next system states from the current system states and applied perturbations. We also use the learned system model in diagnosis, where we reconstruct a possible sequence of the applied perturbations from the recorded system states. We present a case study of an open source system based on ZOOKEEPER and SOLRCLOUD to demonstrate how our techniques reduce the waiting time between the successive perturbation in order to speed up the testing, and improve the diagnosis of failures observed in a deployed system.

Contributions. Our main contributions are summarized as follows:

- We use supervised machine learning to model the system behaviors in response to perturbations, based on recorded observations in a pseudo-distributed setting.
- From the learned model, we predict the next system state given current states and applied perturbations. In a perturbation-based testing framework, accurate prediction helps to shorten the waiting time between the consecutive perturbations (we reduced testing time from 10 hrs to less than 5 hrs).
- From the learned model, we reconstruct a possible sequence of perturbation from a given sequence of observed

system states for diagnosis (we achieve an accuracy of 59%).

- We demonstrate the usefulness of our approach on a case study involving ZOOKEEPER and SOLRCLOUD distributed systems.

The rest of the Chapter is organized as follows. We provide a brief overview of the system used in our case study, i.e., ZOOKEEPER and SOLRCLOUD in Section 8.2. We give an overview of SETSUDŌ testing framework in Section 8.3. We then describe abstraction and modeling of such a system in Section 8.4. In Section 8.5, we describe the learning and prediction using decision tree classifier. In Section 8.6, we explain how we capture system behaviors in response to perturbations. We discuss the implementation aspects involving active and passive monitoring for data collections in Section 8.7. In Section 8.8, we discuss experimental setup and results. Finally, we compare our work with other related efforts in Section 8.9.

8.2 ZOOKEEPER and SOLRCLOUD

In a distributed computing environment, different parts of an application run simultaneously across many machines (nodes), even if the system appears to the client as an opaque cloud that performs the necessary operations. In order to simplify the orchestration of such nodes many services for cluster configuration and coordination have been introduced. Among these services one of the most popular is APACHE ZOOKEEPER [15]. ZOOKEEPER provides reliable distributed coordination that is highly concurrent and suitable mainly for read-heavy access patterns. In particular, it has been adopted for cluster configuration and coordination by many companies as well as for open source enterprise search systems like SOLR [18].

Let us introduce some terminology that will be used in the rest of the Chapter. Note that we are focusing on the open source Apache applications ZOOKEEPER and SOLRCLOUD for the sake of exposition but our approach can be extended and generalized to other scalable distributed systems.

SOLRCLOUD is the name of a set of new distributed capabilities in SOLR, that is an open source enterprise file indexing and search platform from the *Apache LuceneTM project* [2]. SOLRCLOUD is highly reliable, scalable and fault tolerant cluster of SOLR servers. It provides high scalability, fault tolerance (has automated failover and recovery), distributed indexing, and search capabilities. SOLRCLOUD use a leader-replica model and does not use a master node to allocate nodes, leaders and replicas. Instead, it uses ZOOKEEPER to manage these locations, depending on configuration files and schemas. Documents can be sent to any server and ZOOKEEPER will figure it out. ZOOKEEPER is based on a leader-follower model. All write requests from clients are forwarded to a single server, called the *leader*. The rest of the ZOOKEEPER servers, called *followers*, receive message proposals from the leader and agree upon message delivery. The messaging layer takes care of replacing leaders on failures and syncing followers with leaders.

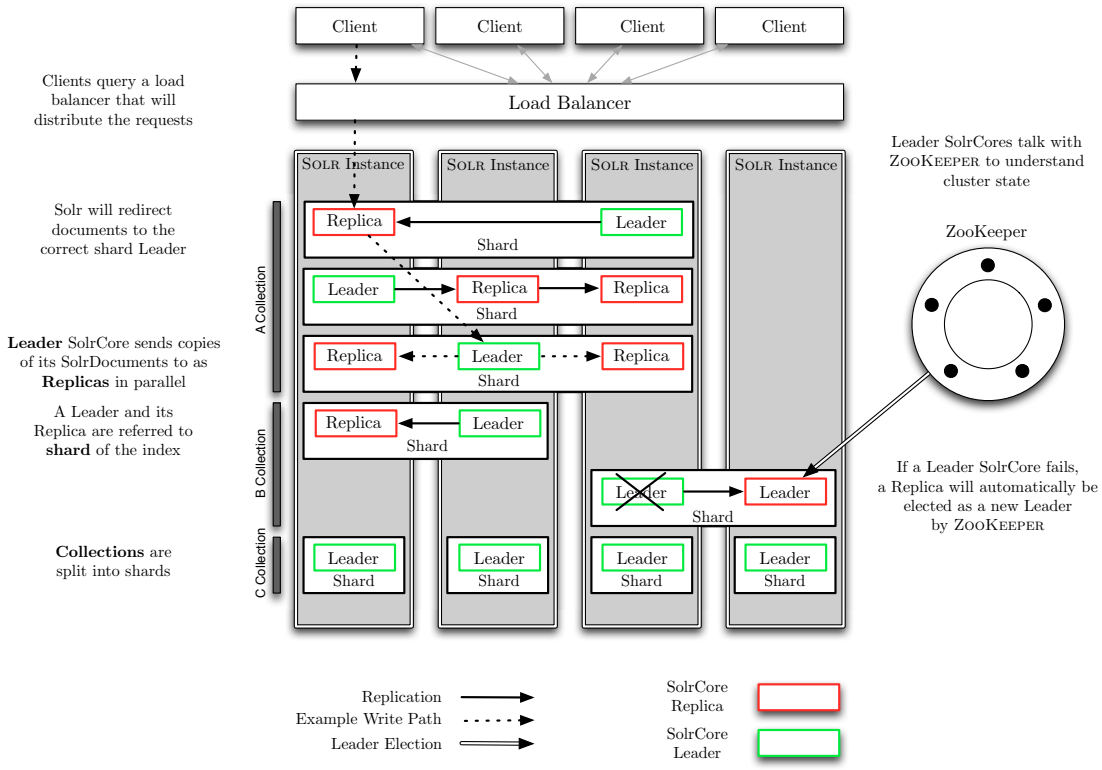


Figure 8.1: SolrCloud Architecture

8.2.1 SolrCloud Architecture

The architecture of SolrCloud is shown in the diagram in Figure 8.1. A complete logical index in a SolrCloud cluster is called a *Collection*. Generally, users will create multiple Collections to separate logical units of data. Collections are generally isolated from one another and do not typically communicate with each other. In our diagram, we have three Collections: A, B and C. These Collections are hosted on a number of Solr Instances (the gray long vertical boxes), each of which is defined as a Solr process contained in a JVM. Let us assume that each Solr instance in this diagram is being run on a separate node in the cluster (however it is also possible to run multiple instances of Solr on a single node).

Collections are made of one or more *SolrCores*. If a Collection is made of or partitioned into many SolrCores, then each partition of the logical index is called a *shard*. In the diagram, the A Collection has three shards, with a number of SolrCores in each of those shards. The B Collection has two shards and the C Collection has four shards. The diagram only shows Collections that are partitioned into shards in order to distribute the index across many nodes. In general, if the index will fit onto a single node it is not necessary to shard the index.

Each shard will contain at least one Leader SolrCore and none to many Replica SolrCores. A Replica is a SolrCore

that contains a copy of a Leader SolrCores index. Replicas are available for failover if their corresponding Leader becomes unavailable. Leaders are responsible for sending out copies of the SolrDocuments to its Replicas, while Replicas and other Leaders are responsible for forwarding the SolrDocument to the proper Leader.

If the index of a Collection is partitioned, then a Leader and its Replicas are defined as a shard. In the diagram we can see that the A Collection is broken into three shards, which have a varying number of SolrCores within them. The first shard of the A Collection has a single Leader and a single Replica. We observe that the Leader sends any SolrDocuments sent to it to its Replica. The second and third shards have three SolrCores each, one Leader and two Replicas. We observe that the client sends a SolrDocument to the Replica of the first shard and the the document gets routed to the proper Leader in the third shard.

The B Collection is split into two shards each with two SolrCores within them. Here the Leader of the second shard has failed. The former Replica of the second shard becomes a Leader automatically. Now any SolrDocuments to be indexed to the second shard of the B Collection will be sent to the new Leader. Admins can bring the dead SolrCore in the meantime. Also note that the Leader (actually all Leader even though it is not depicted in the diagram) communicate with ZooKeeper. ZooKeeper helps manage cluster state, it is aware of which SolrCores are Leaders and which SolrCores have stopped responding. SolrCloud uses this to automatically handle SolrCore failures.

Finally, we observe that the C Collection is split into four shards, each with a single SolrCore within them. Since there are no Replica, if one of those Leaders fails then a portion of the index will be gone.

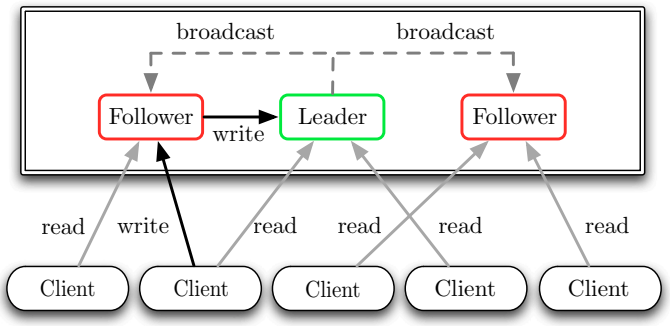


Figure 8.2: ZooKeeper Service. Reads are satisfied by followers, while writes are committed by the leader.

8.2.2 ZooKeeper Architecture

At its core, ZooKeeper is essentially a distributed, hierarchical filesystem comprised of *znodes*. A client is able to create a node, store up to 1MB of data with it, and also associate as many children nodes as they wish. ZooKeeper implements the hierarchical filesystem via an *ensemble* of servers.

Figure 8.2 shows a three server ensemble with multiple clients reading and one client writing. The basic idea is that the filesystem state is replicated on each server in the ensemble. When an ensemble is first started, a leader election is held. During leader election, a leader is elected and the process is complete once a simple majority of followers have synchronized their state with the leader. After leader election is complete, all write requests are routed through the leader, and changes are broadcast to all followers - this is termed atomic broadcast. Once a majority of followers (quorum) have persisted the change, the leader commits the change and notifies the client of a successful update. Because only a majority of followers are required for a successful update, followers can lag the leader which means ZOOKEEPER is an eventually consistent system. Thus, different clients can read information about a given znode and receive a different answer. Every write (i.e. every change to the ZOOKEEPER state) is assigned a globally unique, sequentially ordered identifier called a $ZxID$, or ZOOKEEPER transaction id. This guarantees a global order to all updates in a ZOOKEEPER ensemble. If $ZxID1$ is smaller than $ZxID2$ then $ZxID1$ happened before $ZxID2$. In addition, because all writes go through the leader, write throughput does not scale as more nodes are added.

This leader/follower architecture is not a master/slave setup as the leader is not a single point of failure. If a leader dies, then a new leader election takes place and a new leader is elected. The reason for electing a leader in such a system is to ensure that timestamps assigned to updates are only issued by a single authority. ZOOKEEPER is designed to reduce or eliminate possible race conditions in distributed applications. One feature of ZOOKEEPER's design is that it is intended to serve many more read requests than writes.

All client read requests are served directly from the memory of the server they are connected to, which makes reads very fast. In addition, clients have no knowledge about the server they are connected to and do not know if they are connected to a leader or follower. Because reads are from the in-memory representation of the filesystem, read throughput increases as servers are added to an ensemble.

8.2.3 Operational Specification

ZOOKEEPER supports fault tolerance, as long as a majority of the ensemble are up, the service will be available. Because ZOOKEEPER requires a majority, it is best to use an odd number of nodes. For example, with four nodes ZOOKEEPER can only handle the failure of a single node; if two nodes fail, the remaining two nodes do not constitute a majority. However, with five nodes ZOOKEEPER can handle the failure of two nodes.

On the other hand SOLRCLOUD can continue to serve results without interruption as long as at least one server hosts every shard.

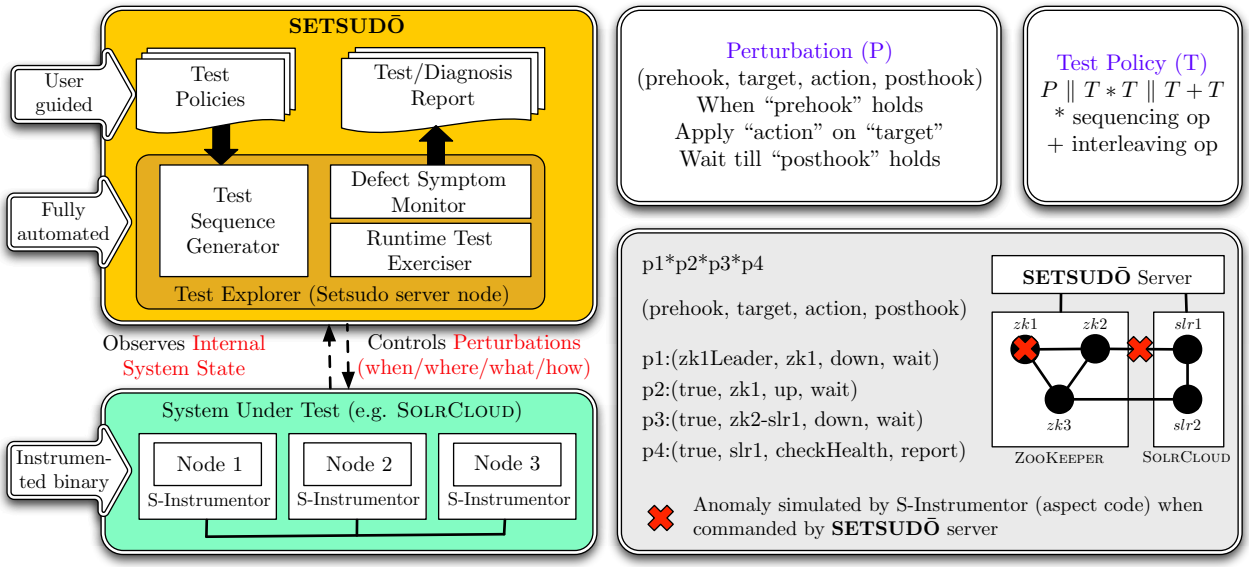


Figure 8.3: SETSUDŌ: Architecture and Flow.

8.3 SETSUDŌ

We discuss briefly the perturbation-based testing platform SETSUDŌ [53] which is targeted to expose deep-rooted robustness defects by applying "perturbation" to the environment of a running (small-scale) cloud system at "suitable" points (internal states). Perturbation corresponds to making a controlled change in the system execution, and environment corresponds to those external factors that are not under the control of the system-under-test. In contrast to traditional black-box testing, internal system states are selectively leveraged to target deep-rooted system defects in small scale tests, thereby avoiding upfront large infrastructure costs.

Testing Platform. The SETSUDŌ platform (shown in Figure 8.3 is designed to: (a) apply perturbations at specific states in the system execution, (b) orchestrate the relative timing between the perturbations, (c) provide a concise way to specify complex scenarios for testing, and (d) provide customization for application-specific testing.

The orchestration is achieved with a SETSUDŌ-server running on a separate node, communicating with a SETSUDŌ-client running on each application server node. The communication between SETSUDŌ server-client allows both observation of system internal states and issuing of perturbation commands. It uses fine-grained control to choose when/where/what/how to perturb the environment. These choices can be guided by testers at a high level using a test policy language framework.

The framework provides a mechanism of exposing the selected internal states, and applying perturbations only when certain state predicates hold on internal states. The machinery of perturbations is implemented using aspect-

oriented programming (AspectJ [5]), and does not require modification of the application code. A perturbation (as shown in the right of the Figure) is a 4-tuple $(prehook, target, action, posthook)$ with the semantics that when "prehook" holds, apply "action" on "target" and wait until "prehook" holds. Specification of a test scenario (at a high level) is supported through a test policy language, where internal-state labels are used to define the timing, target, and action (when, where, and what) of perturbations in a test scenario. A test policy corresponds to sequence ("*") and interleaving ("+") operations on perturbations, and can comprise many sequences of perturbation. One such perturbation $p1 * p2 * p3 * p4$ is shown for SolrCloud service.

Generation of test code from a test scenario is completely automated using *perturbation libraries* (middleware), which produce the desired effect by simulating, and not by actually creating generic or application-specific anomalies. For example, when a "link down" perturbation is to be applied, the framework *simulates* the link being down, rather than actually cutting it off. Also, the link is simulated as being down only at the specified internal state where the prehook holds, and not otherwise. Execution of test code to perturb the environment at runtime is completely automated using *runtime libraries* (middleware) that support weaving in of the test code (aspects) with the application code at runtime, and controlling the execution through the SETSUDŌ-server. This intricate and mostly automated orchestration of perturbations distinguishes this work from fault injection frameworks where a tester has to set up detailed test drivers. Monitoring of defects is also automated by labels in the test policy language. Any check that fails is automatically logged, along with diagnostic information.

SETSUDŌ is customizable and extensible, catering to testing needs at different levels: components (e.g. ZOOKEEPER), applications (e.g. SOLRCLOUD), and domains (e.g. Cloud). It includes a growing base of customized choices (robustness defects, internal system states) through a one-time instrumentation effort for each label in the test policy language. This effort can be amortized over multiple applications on popular components within (or across) domains.

8.4 System Model

We use a state-transition graph to model the observed behavior of an SUT. In particular each state is a *system snapshot* that contains abstract and explicit information about all the nodes in the system. Each edge is labeled with a *perturbation*, that is, an action that we applied on the SUT and the time taken by the SUT to absorb such perturbation completely. Example of perturbations are link/node failures, queries and changes to the index. More details about the perturbations implemented in our framework will be discussed in Section 8.7.3.

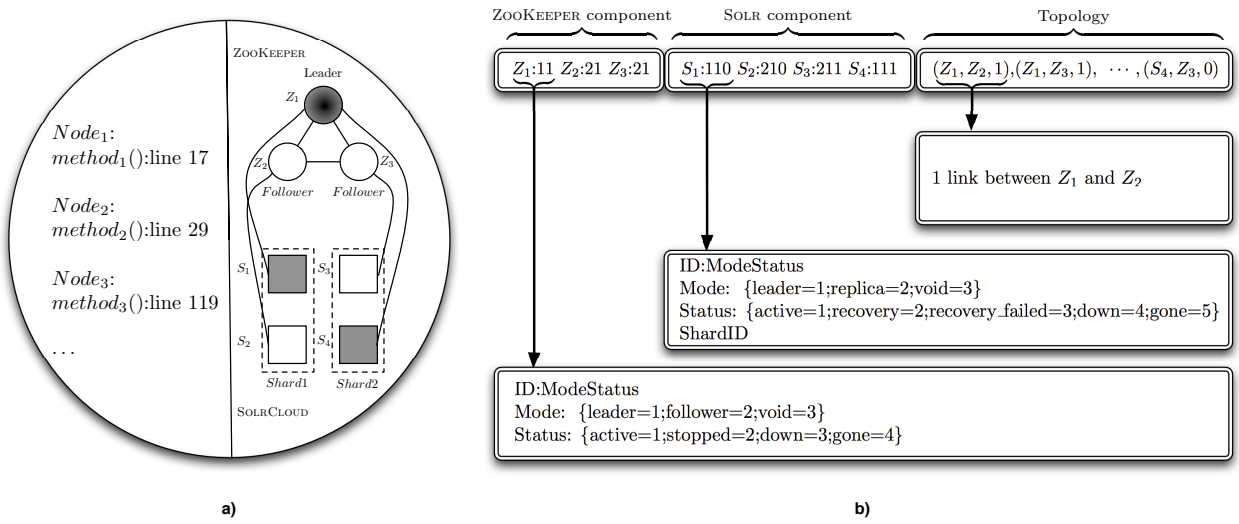


Figure 8.4: a) System Snapshot. b) Abstract System Snapshot

8.4.1 System Snapshots

A system snapshot at a given time comprises recorded information about each node in the SUT. The reader can refer to Figure 8.4a for a simplified version of a system snapshot for a distributed system composed of three ZooKeeper's nodes and four Solr's nodes that control two shards.

With respect to ZooKeeper, for each node we record the following data:

- Serving mode $m \in \{leader, follower, void\}$. Leader and follower do not need additional explanation, *void* is used for nodes that used to be in the system but are currently dead.
- Status $s \in \{active, stopped, down, gone\}$. A node is *active* when it is actively operating in the SUT, replying to queries. It is *stopped* if it replies "*This ZooKeeper instance is currently stopped*" to a query. It is *down* if it replies "*This ZooKeeper instance is not currently serving requests*" to a query. Finally, a status *gone* indicates that the node is dead and out of the system.
- $ZxID$, the global (cluster wide) transaction identifier. A 64-bit number, the upper 32 bits of which are the epoch number (changes when leadership changes) and the lower 32 bits which are the xid proper.
- pid, the process node identifier used by the OS to uniquely identify the process.
- min/avg/max latencies for the node in milliseconds. This details the amount of time it takes for the node to respond to a client request.
- Number of client requests (typically operations) received.

- Number of client packets sent (responses and notifications).

Similarly, we record the following information for each SOLR node:

- Serving mode $m \in \{leader, replica, void\}$. Leader and replica do not need additional explanation, *void* is used for nodes that used to be in the system but are currently dead.
- Status $s \in \{active, recovery, recovery\_failed, down, gone\}$. A node is *active* when it is actively operating in the SUT, it is *down* if it is alive but not responsive and it is gone when the node is dead. The status *recovery* and *recovery_failed* indicate that the node is in the middle of a recovering process (e.g. a node try to get back online after a failure) and the node was not able to do the recovery respectively.
- The shard the node is serving.
- pid, (see above).
- General view, that is the information the node has about the other SOLR nodes in the SUT.

A snapshot also contains information about the topology of the SUT. In particular, for each link in the system we collect the triple $((nodeID,\ port),\ (nodeID',\ port'),\ context)$. That is, we record the information about which nodes and ports are connected and the context at the moment of the link creation. In particular, context is the hashcode of an object containing the location (in terms of method and line number) and the stack when the link was created. Finally, a snapshot contains a reference to which part of the code each node is executing (i.e. the most recent lines in the log files) when the snapshot is taken.

8.4.2 Abstract System Snapshots

In practice, we use an abstraction of the system snapshots. State abstraction is the process of eliminating irrelevant features to reduce the effective state space; such reductions can speed up and improve the quality of the prediction algorithms.

In the abstract system snapshot we considered the serving mode and the status of each ZOOKEEPER and SOLR node, the shard each SOLR node is serving and finally for each pair of nodes we kept count of the number of links between them (see Figure 8.4*b*).

8.4.3 Transient and Steady System States

The systems we deal with are continuously evolving. That is, while a distributed system is running, its nodes evolve and so does the information associated with them, e.g. the role of a node during the leader election. Let us define two abstractions to classify the states of the distributed systems.

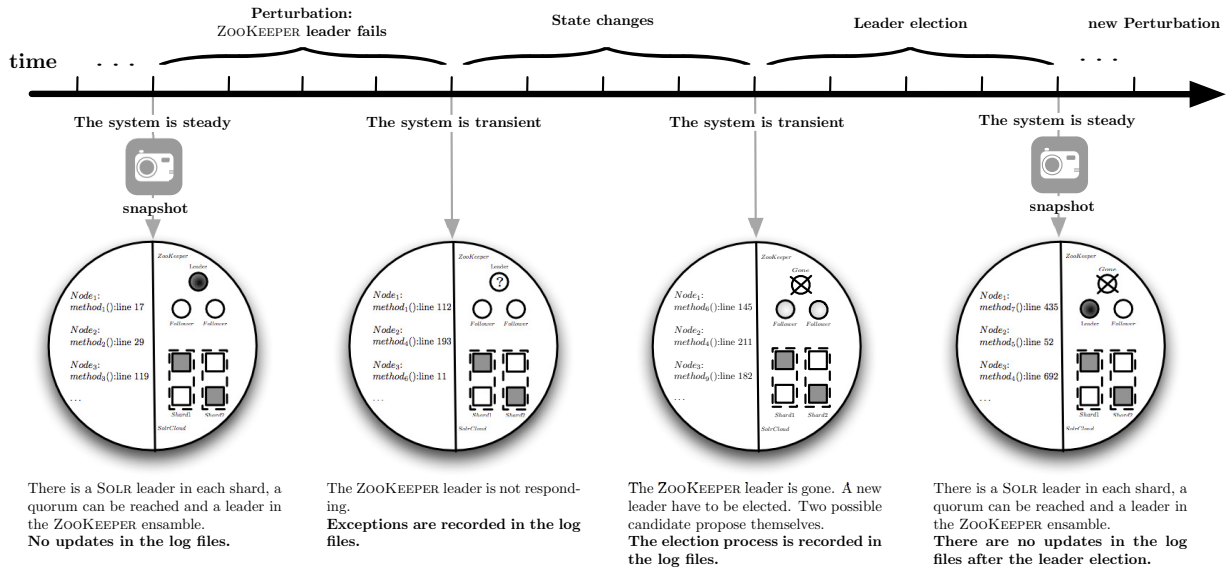


Figure 8.5: Timeline: Perturbation and System Response.

A system is in a *transient* state when there is some pending action in the system and some of the information associated with nodes could change while this action takes place. On the other hand, a system is in a *steady* state when its nodes are stable and there are no pending actions to be digested by the system. In Section 8.7.2 we will provide details about how we detect steady and transient system states in practice.

When a distributed system is in a steady state generally there is low activity in the nodes. For example, ZooKeeper mostly maintains heart beats during steady state. In our model, we will take snapshot of the system only when it is in a steady state (see Figure 8.5) There are two advantages in doing this: (1) we drastically reduce the number of states we have to deal with, and (2) we can focus only on *interesting* configurations of the system. In fact, perturbations are applied on the system when it is not busy in digesting some other change.

Example. In the example shown in Figure 8.5, the SUT is initially in a steady state. There are no recent updates in the log files and the components of the SUT are operating correctly. In particular, ZooKeeper has a working leader and for each shard there is a Solr leader active. Let us apply to this state a perturbation that kills the ZooKeeper leader, let us say Z_1. This leads to the following:

- The other servers try to contact the last leader known not receiving replies and record these failures in the log files. Because the log files are refreshed continuously in this phase, the state of the SUT is transient.
- After some unsuccessful tentatives to reestablish a connection with Z_1, the servers realize that a new leader election is required. All the servers are interested in the result of the election but only some of them are ac-

tively involved in the election process. ZOOKEEPER at the moment implements two leader election algorithms: $LeaderElection$ and $FastLeaderElection$ that are selected depending on the particular configuration of the system. However, the algorithm adopted is not important as long as: (1) the leader has seen the highest $ZxID$ of all the followers, and (2) a quorum of servers have committed to following the leader.

Of these two requirements only the first, the highest $ZxID$ among the followers needs to hold for correct operation. The second requirement, a quorum of followers, just needs to hold with high probability. It will be rechecked later on, so if a failure happens during or after the leader election and quorum is lost, it will recover by abandoning leader activation and running another election. After leader election, a single server will be designated as a leader and start waiting for followers to connect. The rest of the servers will try to connect to the leader. The leader will sync up with followers by sending any proposals they are missing, or if a follower is missing too many proposals, it will send a full snapshot of the state to the follower. All this process is recorded in the log files, making the state of the SUT transient.

- Once a leader is elected, the SUT is operating regularly in all its components (i.e. ZOOKEEPER and SOLR), and there are no further updates in the log files the system is steady and a new snapshot of the SUT is taken.

8.5 Learning and Prediction

To model the system behaviors in response to environmental perturbation, we use *Decision Trees* (DT) [79], a non-parametric supervised learning method that can be used to predict system behaviors by learning simple decision rules inferred from the perturbations and/or system states. For system states, we use abstract system snapshots as shown in Figure 8.4b.

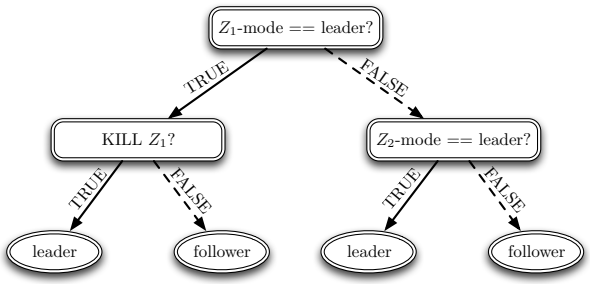


Figure 8.6: DT used for the prediction.

The goal of DT learning is to create a model that predicts the value of an output variables based on several input variables. Figure 8.6 shows a sample decision tree that aims to predict whether the new mode of the Z_2 node will

be *leader* or *follower* given the current state and the applied perturbation. The internal, rectangular nodes in the tree represent tests on input variables and the branches correspond to results of the tests. For example, the root in Figure 8.6 represents the test of whether the Z_1 is leader in the current and the left/right branch is followed if the test is true/false. The leaf nodes of the DT represent the predicted value for the output variable given the outcomes of the test on the path from the root to the leaf. To make a prediction for a test instance where the value of the output variable is unknown, the DT traverses the tree according to the results of the tests in the internal nodes until it reaches a leaf, then it outputs the value of the leaf as the prediction. For the example in Figure 8.6, if the current mode of node Z_1 is *leader*, and the perturbation is *kill* Z_1 then the DT predicts that the new mode of the Z_2 node will be *leader*.

The reasons that governed our selection of DTs for learning are as follows:

- DTs are simple to understand and interpret.
- Cost of predicting is logarithmic in the number of data points used in training.
- DTs use a white box model, i.e., if a given situation is observable in a model, the condition can be explained by Boolean logic.

There are also some known disadvantages.

- DTs suffer from overfitting. Large number of samples are needed to mitigate the effect.
- DTs are generally unstable, and sensitive to variation in the data.

Note that the system that we are modeling are fairly deterministic w.r.t to external perturbations as they react to an environmental perturbation in a specific way as per the implementation of anomaly management. Although machine learning is commonly used to model black-box systems [33], we use it here to model the internal behavior of a system, in order to effectively handle the large state space. We mitigate the two shortcomings of DTs by collecting large sets of training data, and by exploring various perturbation scenarios.

8.5.1 Classification

We use a DecisionTreeClassifier (clf) for classification on a dataset. It takes two arrays: an input array X of size [n_samples, n_features] holding the training samples, and an output array Y of integer values of one feature, size [n_samples], holding the class labels for the training samples. We use $fit(clf, X, Y)$ to carry out fitting, i.e., learning decision rules from the given dataset X, Y. Following fitting, we invoke $predict(clf, T)$ to predict a class label for the output feature, given an input T of size [1,n_features]. For n-outputs (each with possibly different class labels), one can train n-classifiers one for each output.

We propose to use a DT classifier for the following tasks:

- predict the next state given current state and perturbation
- predict a perturbation sequence given a state sequence

The two tasks described above correspond to *testing* and *diagnosis* problems, respectively.

8.5.2 Testing Classifier

Each X sample is comprised of 81 features corresponding to perturbation and current state. A perturbation (component of X) contributes 3 features: `action`, `node1`, and `node2`. An action corresponds to 7 possible values: {*kill, link_down, link_up, query, add, start, delete*}. Nodes, i.e., `node1` and `node2` correspond to 8 possible values: $\{Z_1, Z_2, Z_3, S_1, S_2, S_3, S_4, D\}$, where $Z\langle i\rangle$ corresponds to ZOOKEEPER nodes, and $S\langle i\rangle$ corresponds to SOLR-CLOUD nodes, D corresponds to a don't care node. For actions other than *down/up*, we use `node2` value as *D* (don't care), to denote it is not relevant for that action.

A state (component of X) contributes 78 features corresponding to 3 ZOOKEEPER nodes, 4 SOLR nodes, and their interconnections: `ZooKeeper-mode`, `ZooKeeperstatus`, `Solr-mode`, `Solr-Status`, and `topology`. Each ZOOKEEPER node has 3 modes values {*leader, follower, void*}, and 4 status values {*active, stopped, down, gone*}. Each SOLR node has 3 mode values {*leader, replica, void*}, and 5 status values {*active, recovery, recovery_failed, down, failed*}. The topology corresponds to 21 features (one per node pair), each corresponding to the number of links between a pair of nodes.

For capturing the relationship between "perturbation and current states", and "next states" we require 78 classifiers, one for each next state feature. Let Y_i corresponds to ith feature of the state. Number of class labels for each Y_i sample corresponds to the values of the ith state feature. We invoke $fit(clf_i, X, Y_i)$ to fit each classifier clf_i. The set of classifiers clf_i obtained after fitting operation represents the learned system model. For predicting the next state (with 78 features), we predict the class label (i.e., value) of its ith feature by invoking $predict(clf_i, T)$ where T represent a data point with 81 features (perturbation and current state).

8.5.3 Diagnosis Classifier

For diagnosis purpose, we train the DT classifier with the current and next states as X, and perturbation as Y, respectively.

Each X sample, therefore, comprises of 78*2 = 156 features corresponding to 78 state features. As there are 3 features in a perturbation sample, we require 3 classifiers, one for each feature. Fitting and predicting steps are similar to those described for the Testing Classifier, with the exception that we only consider the cases where the state has

```
1 : void Collect_Data(int n){
2 :   perturbation p;
3 :   int p_applied=0;
4 :   SystemSnapshot s=take_SystemSnapshot();
5 :   SystemSnapshot s1;
6 :
7 :   while(p_applied < p_total){
8 :     p=pick_perturbation(s);
9 :     apply_perturbation(p);
10:     if(!system_gets_steady()){
11:       undo_perturbation(p);
12:       if(!system_gets_steady()){
13:         save_for_inspection(s,p,take_SystemSnapshot(),"U");
14:         reset_system();
15:       }else
16:         save_for_inspection(s,p,take_SystemSnapshot(),"O");
17:       s=take_SystemSnapshot();
18:       continue
19:     }
20:     s1=take_SystemSnapshot();
21:     record_data(Abs(s),p,Abs(s1));
22:     s=s1;
23:     p_applied++;
24:   }
25: }

26: perturbation pick_perturbation(SystemSnapshot s){
27:   perturbation p;
28:   hStatus h=get_health_status(s);
29:   if(h.isHealthful()){
30:     while(true){
31:       p=random(NP, PP);
32:       if(p.isAvailable())
33:         return p;
34:     }
35:   }else{
36:     while(true){
37:       p=random(PP, NP);
38:       if(p.isAvailable())
39:         return p;
40:     }
41:   }
42: }
```

Figure 8.7: Simplified code for the data collection and the perturbation selection.

change as the result of the perturbation. Diagnosing perturbations that do not change the state of the system would be less interesting in practice, and would also be less accurate since there is not enough information to distinguish among the various perturbations that leave the state unchanged.

8.6 Data Collection

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. The data we collect are triples (A, p, A'), where A and A' are abstract system snapshots (taken in a steady state) and p is a perturbation. How we take system snapshots, how we detect steady states and the types of perturbations

implemented are described in Section 8.7. In this section we show how to generate the data and how to select the perturbations to apply to a given steady state.

In Figure 8.7 we report the simplified code for the data collection, generation, and the perturbation selection algorithms. We use a counter $p\_applied$ (line 3) to keep track of the successful perturbation applied to the SUT, that is the samples collected. The collection process stops when the number of perturbation applied is n (line 7). Starting from an initial steady state of the system s, we pick a perturbation p (line 8) and apply it to the SUT (line 9). Notice that in order to inject the perturbations and control the SUT we use a modified version of the SETSUDŌ tool [53]. In particular, we used the methods $apply\_perturbation()$ (line 9) and $undo\_perturbation$ (line 11). The methods $take\_SystemSnapshot()$ and $system\_get\_steady()$ are explained in section 8.7.1 and 8.7.2 respectively.

After the perturbation has been applied there are two possible scenarios:

1. The system reaches a steady state $s1$ (line 20). In such case, we record this transition in our model as an edge between the abstract snapshots of s and $s1$ (line 21). The edge will be labeled with p (we also record the time taken by the transition to take place). We iterate the process of applying a new perturbation to $s1$ (line 22).

2. The system does not reach a steady state (i.e. the system does not stabilize, it remains in a transient state s') and a timeout expires. In this case, we try to *undo* the last perturbation applied to the system.

   If as a result of this undo-ing the perturbation, the system reaches a steady state $s1$, we record (s, p, s') for further inspection (line 16) and we continue the data collection from $s1$ (line 17). We also label the triple (s, p, s') with "O", that stands for Overstressed.

   If even after undo-ing the perturbation, the system state does not become steady we record (s, p, s') for further inspection (line 13). This time we label the triple with "U", that stands for Unhealthful. The meaning of Overstressed and Unhealthful will be clarified soon. We reset the system (line 14) and we re-start the collection process from a new initial state of the SUT (line 17).

As part of data, in addition to the samples collected, we also label the health of the SUT. In Figure 8.8 we show a state transition graph that describes an abstraction of the system behavior. We divided the perturbations in *positive* and *negative* categories (indicated in the figure with PP and NP respectively). In particular, negative perturbations are those that bring links down and kill/stop nodes, and positive perturbations are those that correspond to queries, updates, link up, and node starts (more in Section 8.7.3). We label each edge with the perturbation p applied to the SUT and the time t taken by p to take place. The state of the SUT can be classified based on its health as follows:

- *Healthful*. The SUT is in a Healthful state if all its components are working correctly, that is all the nodes are available and responsive. When the system is in this state, any perturbation $p \in PP \cup NP$ can be applied. But if $p \in NP$, and in particular if p brings down some node of the SUT, then the system will move to a Stressed

state s'. Note that link-down perturbations can also bring the SUT to a stressed state. For example, a node could be up but become isolated from the rest of the network because all the links to/from it are broken.

- *Stressed.* The SUT is in a Stressed state if some of its components are down or out-of-services, e.g. one SOLR replica out-of-service, one ZOOKEEPER node not reachable, etc. However, because the system is fault-tolerant it is still able to operate correctly, possibly with a degraded performance. Also in this state, any perturbation $p \in PP \cup NP$ can be applied to it. The health of the resulting state s' depends on p. s' could be a Healthful state if as a result of the perturbation $p \in PP$ all the components of the system are up and working. Note that we can not expect that system *improve* its health after applying negative perturbations to it. s' could still be in a Stressed state if after that the perturbation is applied the system is still working but one or more nodes are not. Finally, s' could be an Overstressed state if as a result of a perturbation $p \in NP$ the system becomes unavailable.

- *OverStressed.* The SUT is in a OverStressed state if the system is not working. The system is restored by undoing the last perturbation applied to the system. The intuition here is that if the system is not-working, a meaningful perturbation to apply is the one that could reestablish the system.

- *Unhealthful.* The SUT is in an Unhealthful state if as a result of a perturbation the system is not working and even the undo-ing of the last perturbation was not sufficient to restore the system. Generally, this state is indicative of presence of system defects.

This classification has two motivations: for exposing system defects and for selecting perturbations smartly while doing the data collection.

Diagnosis. In terms of diagnosis, the Unhealthful state is likely beyond the SUT operational specification, and is the most interesting from the tester point of view as it could actually hide defects. Because, even undo-ing the perturbation that drove to an unresponsive state, the system is unable to recover. An additional investigation should be done in order to confirm that or explain why the system become unresponsive.

Perturbations selection. The perturbations to apply in sequences are selected dynamically depending on our knowledge of the last state of the SUT. The health classification is useful to avoid a completely random perturbations selection and to realize quality testing. A perturbation may not be applicable at a given state if it is redundant or will not lead to interesting scenarios. We derive an *available* set (a subset) by selectively filtering out those non-applicable perturbations.

In Figure 8.7 we report the simplified code for the perturbation selection. Given the health of a state s (line 28) we proceed as follow:

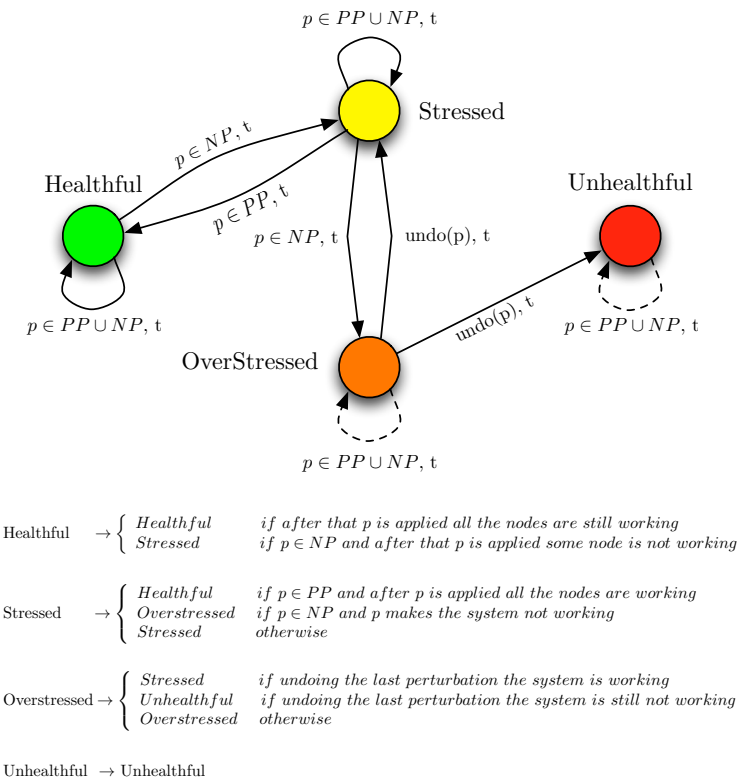


Figure 8.8: Abstraction of System Behavior. PP and NP indicate positive and negative perturbations respectively, t indicates the time took by the perturbation p to be digested by the system.

- If s is an Healthful state, we would like to apply a negative perturbation $p \in NP$ in order to exercise the SUT (line 31). This is done to reduce the non-applicability of a perturbation. For example, if a link between two nodes does not exist, a perturbation p that breaks such link is redundant and then essentially is ineffective. We restrict the selection of perturbations to such an available set (line 32).

- If s is an Stressed state, we restrict the available set to positive transitions to reduce the likelihood of inconsequential perturbation (line 37). For example, if a node is down, all the perturbations that try to bring down links to/from such node are no longer applicable.

- If s is a OverStressed or Unhealthful state theoretically any perturbation is applicable (dotted self-loops in Figure 8.8). However, with a system not working many perturbations would be just meaningless. A reasonable way to proceed is to apply a perturbation to undo the previous perturbation that drove the SUT to the OverStressed/Unhealthful state. The management of these cases is done in the method $Collect\_Data()$ (line 10-19).

8.7 Implementation

In this section we show how we take system snapshots (implemented in the method $take\_SystemSnapshot()$), how we detect if a system is in a steady state (implemented in the method $system\_get\_steady()$), and finally we discuss the kinds of perturbation we implemented.

8.7.1 Take System Snapshots

In order to take a snapshot of the SUT, different techniques are applied. In particular we use a combination of instrumentation, *active* and *passive* queries to the system. The instrumentation is realized using AspectJ [5] to intercept system calls performing network I/O. Every time a socket between two peers is created (or closed), the information about the peers, the ports used and the context when the creation occurred are recorded. Context is the hashcode of an object containing the location (in terms of method and line number) and the stack when the socket was created. However, our knowledge of the status of the links may not be accurate. For example, if a node crashes then there are no records about the deletion of the sockets associated with it. In order to obtain a precise topology, we query the OS using the command $netstat$ to check if each socket recorded as existing still exists. ($netstat$ is a commandline tool used to display very detailed information about how a node is communicating with other nodes or network devices.) We benefit from other OS commands:

- ps. Command used to displays the currently-running processes on a node. In particular, we used ps to obtain the pids of the processes in the system.
- nc. nc (or $netcat$) is designed to be a dependable back-end that can be used directly or easily driven by other programs and scripts. We used nc to interact with ZOOKEEPER. It is in fact a fairly common habit of the application server developers to provide commands that can be issued through nc (or $telnet$) at the client port and that provide information about the application. For example, ZOOKEEPER responds to a small set of commands. Each command is composed of four letters.
  - **dump**. Lists the outstanding sessions and ephemeral nodes. This only works on the leader.
  - **envi**. Prints details about serving environment
  - **kill**. Shuts down the server. This must be issued from the node the ZOOKEEPER server is running on.
  - **reqs**. Lists outstanding requests
  - **ruok**. Tests if server is running in a non-error state. The server will respond with $imok$ if it is running. Otherwise it will not respond at all.
  - **srst**. Resets statistics returned by stat command.

– **stat**. Lists statistics about performance and connected clients.

We used the command $stat$ to obtain the serving mode, the status, the $ZxID$ and stats on latency/packets of each node.

- $wget$. $wget$ is a command that retrieves content from web servers. In particular, we issue queries to ZOOKEEPER through $wget$ to obtain the *clusterstate.json* object associated with each SOLR node. $json$ or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. From a given *clusterstate* associated with a SOLR node we extracted the serving mode, the status and the particular shard served by the node. Again, information may not be accurate and we use additional cross checks to refine such information. For example, if a $Solr$ leader suddenly crashes the *clusterstate* returned by ZOOKEEPER may not reflect such changes and wrongly claim that the node is still the leader. We used ps to verify that the node was alive before to trust the information contained in the *clusterstate*.

Finally, we analyze the log files of each node to collect the context at the time when the snapshot is taken (last location recorded in the log files). This context gives us useful insights regarding behavioral information such as if the node was involved in a leader election, a load balancing, or a recovery.

Note that all the commands that we used are fairly common among various OS platforms. In order to test other distributed systems, specific wrappers may be required. However, snapshots can be taken on any system using available commands to query the system about its status.

8.7.2 Detecting Steady State

In order to detect when the system is in a steady state we apply a two phase analysis:

- No-changes. We monitor the log files and record when the last update on such files was done. When a threshold is reached since last update we move to the second phase.

- Regularity. Once we have checked in the first phase that the nodes are not updating information in the log files we check the regularity of the system components. In particular, we check if the ZOOKEEPER component is working (i.e. a leader exists and reply to queries). Then we check if for each shard there is a SOLR leader active. If these regularity checks fail we go back to the No-changes phase.

This two-phases analysis is iterated until a steady state is detected or a timeout expires. In the case of timeout we report a transient state. In our experiments, we used a timeout of 1 minute.

8.7.3 Types of Perturbation Implemented

We can apply different kinds of perturbations like network failures, network congestions, node crashes, node suspension, etc. We classified the perturbations that we implemented as negative or positive, depending on the expected effects of such perturbations on the system.

- **Negative Perturbations.** These correspond to perturbations that bring a Healthful state to a Stressed state or a Stressed state to Unhealthful or Overstressed state. These include failures injections in network/node/disk, and delays in response times. To fail a network connection between two nodes, instead of allowing the system calls that perform network I/O between the two nodes to execute and return value successfully, the instrumentation forces them to throw I/O exceptions and return unsuccessfully. Note that the network failures can be inside the ZOOKEEPER component, inside the SOLRCLOUD component or a combination of them. We can also simulate node crashes or CPU overloads, which are simulated by killing or temporarily suspending the node process, respectively.

- **Positive Perturbations.** These correspond to perturbations that bring a Stressed/Healthful state to Healthful state or Overstressed to Stressed/Healthful state. These include undo-ing the effect of negative perturbation or involve application query/update. We also implemented application specific perturbations for workload requests, such as add files to the index, remove files from the index, query the nodes for some files. These perturbations are obtained using $wget$ on the specific nodes. *Undo* of a previous perturbation is classified as positive. It could be applied if the system is not stabilizing as described in Section 8.6, or simply if the perturbation we are applying is a network recovery, node restart, node resume. In case of a network recovery the instrumentation will stop failing the system calls that perform network I/O for the connection with exceptions, and would allow the calls to execute as they would have without its intervention. Similarly, to undo a node crash, we re-start the process for the node. Finally to undo a node suspend, we send a SIGCONT to the process.

8.8 Experiments

8.8.1 Data Collection

We ran all experiments on an Apple MacBook Pro with 2.4 Ghz Intel Core i5 processors and 8GB of memory, running OS X 10.7.5. We used for our experiments 4 SOLR servers (release 4.3.0) and 3 ZOOKEEPER servers (release 3.4.5) running in standalone mode. The data indexed in SOLR are those contained in the folder $example$ that come with the release of SOLR.

The test setup for each data collection is as follows: as the servers starts and the system state is steady, (detected

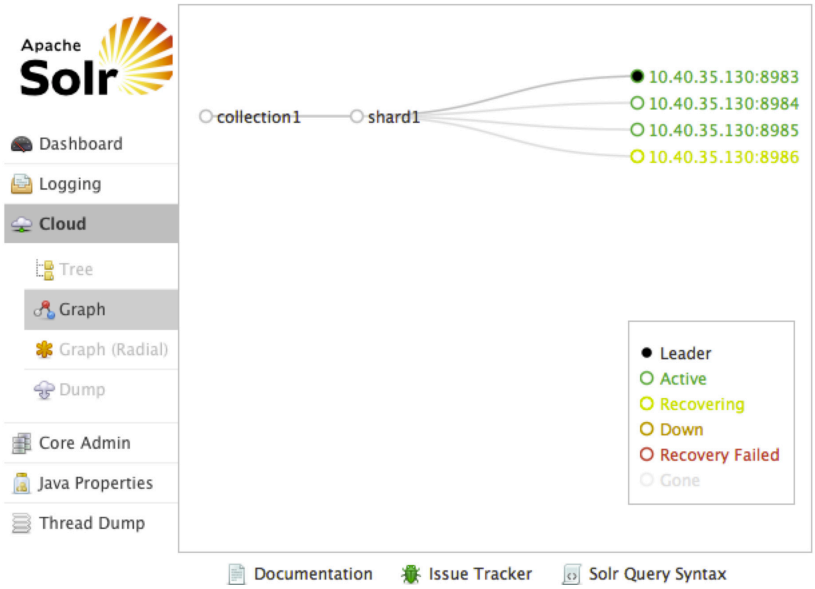


Figure 8.9: Graph representation (part of the SOLR release) of the SOLR components resulting from a query to the system.

as described in Section 8.7.2) a snapshot of the system is taken. In order to take the snapshot we used a combination of instrumentation, active and passive queries to the system. ZOOKEEPER and SOLR releases include commands to query the status of the servers. In Figure 8.9, we show the graph representation (part of the SOLR release) of the SOLR components resulting from a query to the system. We adapted these commands in order to collect the information relevant for the system snapshots.

After the snapshot is taken, our algorithm selects and applies a perturbation to the SUT and then waits for the next steady state. This process iterates until the system is in a Unhealthful state or 100 perturbations are applied to the SUT. In both cases, we reset the SUT and we re-start the data collection. We generated 65 sequences of perturbations and applied a total of 3570 perturbations: 1651 of which are Negative and the remaining 1919 are Positive. On average, our framework waited 79 secs between consecutive perturbations. It includes the time to undo a perturbation and to detect an unresponsive state of the SUT that are the most expensive situations given the timeouts. We require 336 undo perturbations, 303 of which successfully brought the SUT from Overstressed to Healthful/Stressed states. These numbers indicate that the Negative and the Positive perturbations were well balanced, forcing the SUT to transition between different non-Unhealthful states. Moreover, if we exclude the nine sequences where 100 perturbations were applied, we found that in 54 states the SUT was not able to recover, and was in Unhealthful state. In Figure 8.10 we report for each data collection the number of perturbations applied to the SUT.

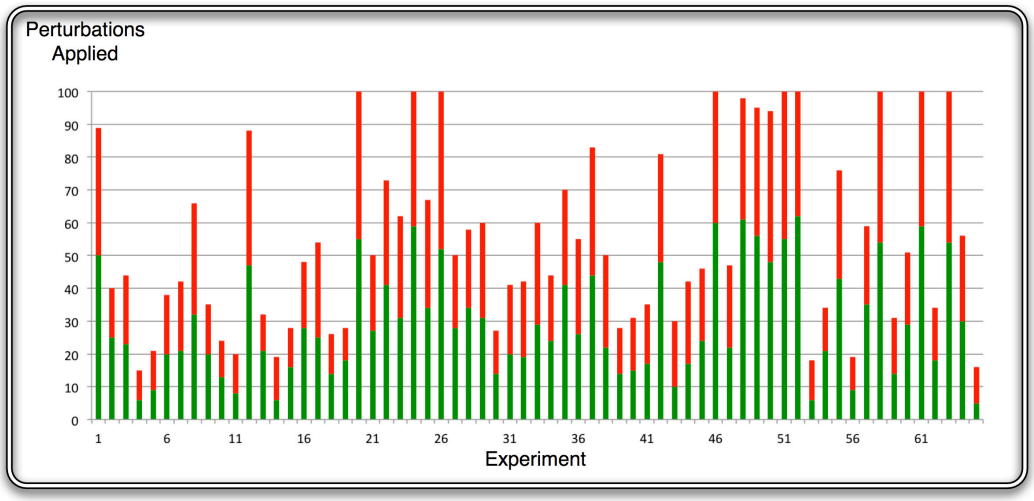


Figure 8.10: Perturbations applied to the SUT for each data collection. The red and the green bars indicate the Negative and the Positive perturbations respectively.

8.8.2 Learning

We used an open source machine learning kit `scikit` [6] for learning system models and predicting system behaviors. We used the default setting for the classifier with `gini` criterion. We used 90% of collected samples for training the classifier, and the remaining 10% for evaluating the classifier. We used one hot-encoding for each feature. In other words, if a feature has n values, we used a Boolean vector of size n with only the ith position (corresponding to the ith value) set to 1, and the remaining positions set to 0. This encoding increases the number of features, but leads to better decision rules in practice.

For 3570 samples, our Testing Classifier gives about 80% accuracy in the prediction, and Diagnosis Classifier

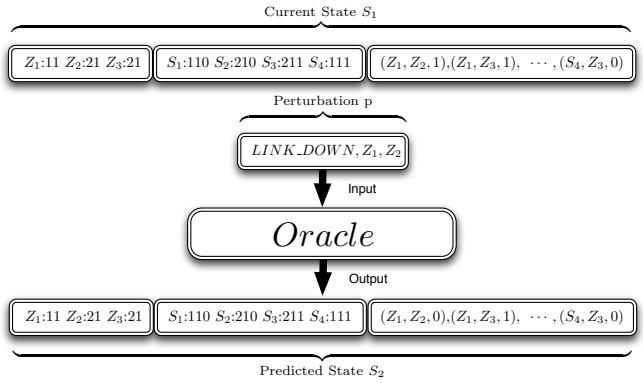


Figure 8.11: Testing Oracle Flow.

gives about 59% accuracy. We plan to extend our experiments to a larger number of sequences in order to improve our prediction accuracy. Other possible directions are discussed in Section 8.8.4.

8.8.3 Evaluation of Testing Classifier

In order to evaluate the testing classifier in terms of time saved during the testing, we use an *Oracle*. The Oracle, implemented in Python on top of the `scikit` [6] tool is reported in Figure 8.11. The Oracle, takes in input a steady state S_s, a perturbation p and return a predicted steady state S'_s.

Given the current steady state of the SUT S_s, we apply a perturbation p to it and we take snapshots of the SUT while the system is evolving. Let us indicate with S' the current snapshot taken. We stop when a steady state is reached, that is S' is steady. The idea is to compare the state S' with the state S'_s returned by the call $Oracle_t(S_s, p)$. Let us define T as the time took by the SUT to move from S to S', with S' steady, given a perturbation p applied to S. Moreover let us indicate with t the time from when the perturbation p is applied up to when a state match between S' and S'_s is found. Notice that S' could be either steady or transient.

While comparing S' and S'_s there are three possible outcome:

1. $S' = S'_s$ and S' is steady. In this case we have an accurate prediction and we have a time saving of $(T - t)$.
2. $S' = S'_s$ with S' transient but when the SUT reaches a steady state S''_s, $S''_s \neq S'_s$. Intuitively, at some point the SUT was in a transient state S' that was matching the predicted state S'_s. However, once p was fully digested and the SUT reached the steady state S''_s, then there is no match between S''_s and S'_s. In this case we have an accuracy problem. This condition is aliasing due to the state abstraction.
3. $S' \neq S'_s$ with S' steady or transient. This condition holds when there was never a match between the predicted state and the state of the SUT during the whole time that p was digested. In this case we have a missed prediction and we have 0 time saving.

We let the SUT run for ten hours where we iterated over applying perturbations. For each perturbation, we waited 2 minutes to allow the SUT to digest the perturbation applied (similar to what testing tools such as SETSUDŌ do). We abstracted out topology information in the evaluation. Out of the 300 perturbations applied, we observed:

- 235 times, case (1), with an average saving time of 76 secs. In other 21 cases, there was an initial mismatch between the state predicted by the oracle and the actual state of the SUT. A match was found after 69 secs (in average), i.e. a saving time of 51 secs.
- 0 times, case (2).
- 44 times, case (3) where the oracle missed the prediction, resulting in no time saved.

In total, the time saved was 315 mins, bringing down the testing time from 10 hrs to less than 5 hrs. That means a saving of 53% of time that tools as SETSUDŌ would take for running this test.

The testing classifier could be improved by using more powerful learning algorithms and by using features derived from the input variables that would make learning easier. Also, the classification is done using one classifier for each output. It follows that invalid states could be returned (e.g. a state with multiple leaders). One can impose constraints on the returned states to make sure they are valid. Weighted sat solvers could be used to find a returned state that best matches the predictions of the classifiers while satisfying the validity constraints (possible improvements are discussed in the next Section).

8.8.4 Future Improvements

In this Section we describe some of the improvements that can be done in order to achieve a higher prediction accuracy.

- The testing classifier could be improved by using more powerful learning algorithms (e.g. bagged or boosted decision trees) and by using features derived from the input variables that would make learning easier (e.g. "killed node is a leader (true/false)" or "number of active nodes"). Also, the classification is done using 78 independent classifiers, one for each output . The drawback of this approach is that the classifiers could return invalid states (e.g. a state with multiple leaders). One can impose constraints on the returned states to make sure they are valid. Weighted sat solvers could be used to find a returned state that best matches the predictions of the classifiers while satisfying the validity constraints.

- For the diagnosis we train the DT classifier with the current and the next states as X, and perturbation as Y, respectively. An interesting direction to pursue would be to consider two prior states and the next states as X. We expect that relating the two prior states will improve the quality of the prediction (improve accuracy and reduce misprediction).

- For the evaluation of the testing classifier, we fixed a timeout of 2 minutes per each perturbation to get absorbed by the SUT. It was not possible to use the dynamic procedure based on the log updates (described in Section 8.7.2).

  In particular, the evaluation required taking snapshots of the system *continuously* (we took snapshots every 10 seconds) in order to compare the current state of the SUT with the one returned by the oracle. This resulted in changing of the log files due to active queries in recording the snapshots, which affected the stability check (i.e. No-changes phases was not reached in our experiments). Exploiting other stability criteria would be an interesting problem to solve.

- Finally, it would be interesting to extend our experiments to a larger number of sequences and do the prediction using topology information also.

8.9 Related Work

Stress testing has been extensively exploited by different tools as HP's LoadRunner and QTP [12], Apache JMeter [13], and Selenium [17]. The goal is to test systems under heavy load conditions to check robustness, availability, tolerance, error handling, etc. However, these frameworks have several inherent limitations. In particular, manual effort is required to generate test scenarios. Some tools are unaware of SUT internals, resulting inadequate in exposing particular events and orderings. They have high-cost in terms in setting up a large-scale test system, and finally they have limited coverage.

On the other hand our framework is completely automatic. The perturbations are selected by the internal state of the SUT. We require low-cost infrastructure (small-scale) and we avoid meaningless tests. Finally we are able to explore complex scenarios.

In the context of *cloud testing*, some recent efforts [47, 54] have focused on testing failure recovery.In FATE and DESTINI [47], failures are systematically injected in disk/node/link in various combinations, followed by checks to see if the system tolerates these failures and behaves as expected, based on user-provided specification.

In contrast, SETSUDŌ [53], provides abstractions of system-specific states that can be used by the testers to specify failures (and other more general perturbations), and to decide the granularity at which we want to distinguish between failure scenarios unlike in [47].

There are other fault injection frameworks based on systematic exploration via model checking, such as EXPLODE [104] and FiSC [105], that explore thousands of program states and inject crashes at every unique program state. MoDist [103] intercepts various OS operations during execution, and exhaustively tests against all possible orderings of those operations and all possible failures that can occur during those operations. DeMeter [48] reduces the number of orderings and failure sequences that a model checker like MoDist has to explore, but the reduction might still not be enough for a tester with constrained resources.

In comparison, our approach (like the SETSUDŌ framework) exposes abstractions of internal states of an SUT. In SETSUDŌ [53] such abstractions were used for finely controlled exploration of system executions. Our learning-based framework presented here facilitates exploration of the perturbation space by providing automated utilities for selecting, scheduling and executing the perturbations.

Machine learning methods are very promising due to their ability to manage with uncertainty. Dietterich et al. identified in 2008 software testing as one of the most challenging domains for machine learning over the successive

ten years [35]. At the same time, Namin and Sridharan [72] recently recognize that, despite the potential of learning methods, their usage (especially referring to Bayesian methods) in software testing is still in its early stages. Nevertheless, some relevant examples of learning methods applied to software testing have appeared in the literature. In [26], authors propose to adopt decision trees to improve test suites in category-partition testing in what they called the MELAB (MachinE Learning based refinement of BlAck-box test specification) process. The same author then described some other experience in applying machine learning in testing-related problems [25], e.g., in debugging/fault localization, in fault prediction for aiding risk-driven testing, and in automatic test oracles identification.

Some previous work such a such as [46, 33] proposed for detecting performance anomalies using machine learning approaches. In [33], authors uses unsupervised learning method for capturing system behaviors such resource usage (CPU, memory, network and disk I/O) and predicting performance anomalies for a running IasS cloud system. In [46], authors used Markov models with naive Bayesian classification to predict performance anomalies. In contrast, we use machine learning to improve the performance of testing and quality of diagnosis.

Several previous works, (for example Peled et al. [76], Groce et al. [45] and Raffelt et al. [77] have considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Sankaranarayanan et al. used a combination of testing and decision tree classifier to automatically infer data preconditions for library functions [81]. Within the model checking community the verification approach known as counterexample guided abstraction refinement (CEGAR) also combines learning and model checking, (see e.g. Clarke et al. [31]).

Chapter 9

Conclusion

The prediction algorithms that we propose in this thesis are a promising step towards testing concurrent programs efficiently and effectively. The algorithms are based on a combination of static analysis and logical constraint solving.

We implemented the proposed algorithms in a tool, PENELOPE, and applied to a set of 18 concurrent programs exploring prediction of null-pointer exceptions, data-races, deadlocks and atomicity-violations. The results of these investigations have shown the efficiency and effectiveness of our proposal. PENELOPE has been released and has attracted the attention of several research groups which are using it in their research projects.

We ran the programs under the test harness several times, and did not find (almost) any of the reported bugs in any of these benchmarks by merely running tests randomly. It was clear that a more focused approach is absolutely necessary in finding errors on these benchmarks. Despite its small selection of schedules to test, PENELOPE was able to identify bugs in these programs.

We proposed an aggressive pruning of executions using static analysis based on vector-clocks that identifies a small segment of the observed run on which the prediction effort can be focused. This greatly improves the scalability of using sophisticated logic solvers. Pruning of executions does not affect feasibility of the runs, but may reduce the number of runs predicted. However, we show that in practice no additional errors were found without pruning.

We formulated a new prediction model at the shared communication level that allows some leeway so that the prediction algorithm can predict runs with small *deviations* from the observed run which could be interesting runs; this makes the class of predicted runs larger at the expense of possibly making them *infeasible*, though in practice we found the majority of the predicted runs to be feasible.

The runs predicted using the techniques proposed may be infeasible, or may be feasible and yet not cause any error. We mitigate this by a re-execution engine that executes predicted schedules accurately to check if an error actually occurs. Errors reported hence are always real and hence we incur no false positives.

Overall, the runtime overhead in precisely scheduling the alternate executions is not prohibitively high, and is in fact very minimal in most examples. This is despite the large number of context-switches that are being exercised in some of our benchmarks.

The extendibility and the applicability of our techniques to a variety of problems in predicting alternate interleaving

in concurrent program analysis (e.g. violations of Java2SDK library properties) pose an interesting challenge which demands attention.

Additionally, in the context of testing scalable distributed systems, we presented an approach based on perturbation-based learning to model system behaviors using Decision Trees. Using the learnt model, we predict the next state with a goal to reduce the testing time. We also use similar techniques to construct the perturbation history from given observed states.

We presented a case study on an open source Apache project, and showed the usefulness of the approach (we reduced the testing time from 10 hrs to less than 5 hrs, and achieved a diagnosis prediction accuracy of 59%). More investigations will be required to improve the accuracy, but we strongly believe that the proposed approach is a serious candidate to achieve qualitative testing in the context of distributed systems.

References

[1] Microsoft Poirot.
http://research.microsoft.com/en-us/projects/poirot.

[2] Apache SOLRCLOUD
http://wiki.apache.org/solr/solrcloud/.

[3] DBCP.
http://commons.apache.org/dbcp.

[4] Java Grande Benchmark suite.
http://www.javagrande.org/.

[5] Apache AspectJ
http://eclipse.org/aspectj/.

[6] Scikit-learn: Machine learning in python.
http://scikit-learn.org/.

[7] BCEL.
http://jakarta.apache.org/bcel.

[8] Apache Common Project.
http://commons.apache.org.

[9] HEDC.
http://www.hedc.ethz.ch.

[10] Colt.
http://acs.lbl.gov/~hoschek/colt.

[11] Apache FTP.
http://mina.apache.org/ftpserver.

[12] HP - enterprise software.
http://www8.hp.com/us/en/software/enterprise-software.html.

[13] Apache JMeter.
http://jmeter.apache.org/.

[14] Java Sun.
http://java.sun.com.

[15] Apache ZOOKEEPER
http://http://zookeeper.apache.org.

[16] IBM ConTest.
http://www.alphaworks.ibm.com/tech/contest.

[17] Selenium - Web Browser Automation.
http://docs.seleniumhq.org/.

[18] Apache SOLR
http://lucene.apache.org/solr/.

[19] Weblech.
http://weblech.sourceforge.net.

[20] Rahul Agarwal, Saddek Bensalem, Eitan Farchi, Klaus Havelund, Yarden Nir-Buchbinder, Scott D. Stoller, Shmuel Ur, and Liqiang Wang. Detection of deadlock potentials in multi-threaded programs. *IBM Journal of Research and Development*, 54(5), September/October 2010.

[21] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and runtime f. In *In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*, pages 191–207. Springer-Verlag, 2005.

[22] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 98–109, New York, NY, USA, 2005. ACM.

[23] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD*, pages 41–50, 2006.

[24] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*, pages 208–223, 2005.

[25] Lionel C. Briand. Novel applications of machine learning in software testing. In *QSIC*, pages 3–10, 2008.

[26] Lionel C. Briand, Yvan Labiche, Zaheer Bawar, and Nadia Traldi Spido. Using machine learning to refine category-partition test specifications and test suites. *Inf. Softw. Technol.*, 51(11):1551–1564, 2009.

[27] Yan Cai and W.K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *ICSE*, 2012.

[28] Feng Chen, Azadeh Farzan, José Meseguer, and Grigore Rosu. Formal analysis of java programs in javafan. In *CAV*, pages 501–505, 2004.

[29] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 240–253, 2007.

[30] Feng Chen, Traian Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 221–230, New York, NY, USA, 2008. ACM.

[31] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Strichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *CAV*, pages 265–279, 2002.

[32] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[33] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. UBL: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, pages 191–200, 2012.

[34] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Softw., Pract. Exper.*, 29(7):577–603, 1999.

[35] Thomas G. Dietterich, Pedro Domingos, Lise Getoor, Stephen Muggleton, and Prasad Tadepalli. Structured machine learning: the next ten years. *Machine Learning*, 73(1):3–23, 2008.

[36] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 411–422, New York, NY, USA, 2011. ACM.

[37] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252, October 2003.

[38] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 52–65, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 155–169, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 248–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Azadeh Farzan, M. Parthasarathy, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *SIGSOFT FSE*, page 47, 2012.

[42] C. Flanagan and S. N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[43] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, 2003.

[44] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[45] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.

[46] Xiaohui Gu and Haixun Wang. Online anomaly prediction for robust cluster systems. In *ICDE*, pages 1000–1011, 2009.

[47] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 18–18. USENIX Association, 2011.

[48] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278. ACM, 2011.

[49] J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *VMCAI*, pages 175–190, 2004.

[50] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.

[51] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 144–154, New York, NY, USA, 2011. ACM.

[52] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE)*. IEEE, 2012. to appear.

[53] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N.Papakonstantinou. SETSUDO: Perturbation-based testing framework for scalable distributed system. In *TRIOS: Conference on Timely Results in Operating Systems*, 2013.

[54] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. *SIGPLAN Not.*, 46(10):171–188, October 2011.

[55] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *SIGSOFT FSE*, pages 327–336, 2010.

[56] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120, New York, NY, USA, 2009. ACM.

[57] Horatiu Jula, Daniel M. Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 295–308, Berkeley, CA, USA, 2008.

[58] Vineet Kahlon, Franjo Ivanăić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 505–518, Berlin, Heidelberg, 2005. Springer-Verlag.

[59] Vineet Kahlon and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag.

[60] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 509–524, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, NY, USA, 2010. ACM.

[62] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[63] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[64] K. Li, P. Joshi, A. Gupta, and M. Ganai. Reprolite: A lightweight tool to quickly reproduce hard system bugs. In *Under submission*, 2014.

[65] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[66] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[67] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 37–48, New York, NY, USA, 2006. ACM.

[68] Zhi Da Luo, Raja Das, and Yao Qi. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *ICST*, pages 309–318, 2011.

[69] Patrick Meredith and Grigore Roşu. Runtime verification with the RV system. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 136–152, Berlin, Heidelberg, 2010. Springer-Verlag.

[70] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[71] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[72] Akbar Siami Namin and Mohan Sridharan. Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 349–354, New York, NY, USA, 2010. ACM.

[73] Christos Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., New York, NY, USA, 1986.

[74] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 135–145, New York, NY, USA, 2008. ACM.

[75] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 25–36, New York, NY, USA, 2009. ACM.

[76] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, 2001.

[77] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. In *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'07, pages 136–152. Springer-Verlag, 2008.

[78] Zvonimir Rakamarić. Storm: static unit checking of concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 519–520, New York, NY, USA, 2010. ACM.

[79] L. Rokach and P. Maimon. *Data mining with decision trees: theory and application*. World Scientific Pub Co Inc, 2008.

[80] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[81] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, pages 295–306, 2008.

[82] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), 1997.

[83] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *STTT*, 8(3):248–260, 2006.

[84] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[85] T.F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. Technical report, University of Illinois at Urbana-Champaign, October 2011.

[86] Vivek K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *APSEC*, pages 361–368, 2008.

[87] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 47–56, New York, NY, USA, 2010. ACM.

[88] Nishant Sinha and Chao Wang. On interference abstractions. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 423–434, New York, NY, USA, 2011. ACM.

[89] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[90] Francesco Sorrentino. PickLock: A deadlock prediction approach under nested locking. *Under Review.*, 2014.

[91] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[92] Francesco Sorrentino, Malay K. Ganai, Alexandru Niculescu-Mizil, and Aarti Gupta. Improving testing and diagnosis of scalable distributed systems through perturbation-based learning. *Under Review.*, 2014.

[93] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[94] C. von Praun and T. R. Gross. Object race detection. *SIGPLAN Not.*, 36(11):70–82, 2001.

[95] Christoph von Praun. Detecting synchronization defects in multi-threaded object-oriented programs. In *Ph.D. Thesis*, Swiss Federal Institute of Technology, Zurich, 2004.

[96] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[97] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 328–342, Berlin, Heidelberg, 2010. Springer-Verlag.

[98] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 137–146, New York, NY, USA, 2006. ACM.

[99] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006.

[100] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, pages 281–294, 2008.

[101] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *ECOOP*, pages 602–629, 2005.

[102] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[103] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228. USENIX Association, 2009.

[104] Junfeng Yang, Can Sar, and Dawson Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 10–10. USENIX Association, 2006.

[105] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, November 2006.

[106] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[107] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 251–264, New York, NY, USA, 2011. ACM.

[108] Wei Zhang, Chong Sun, and Shan Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 179–192, New York, NY, USA, 2010. ACM.